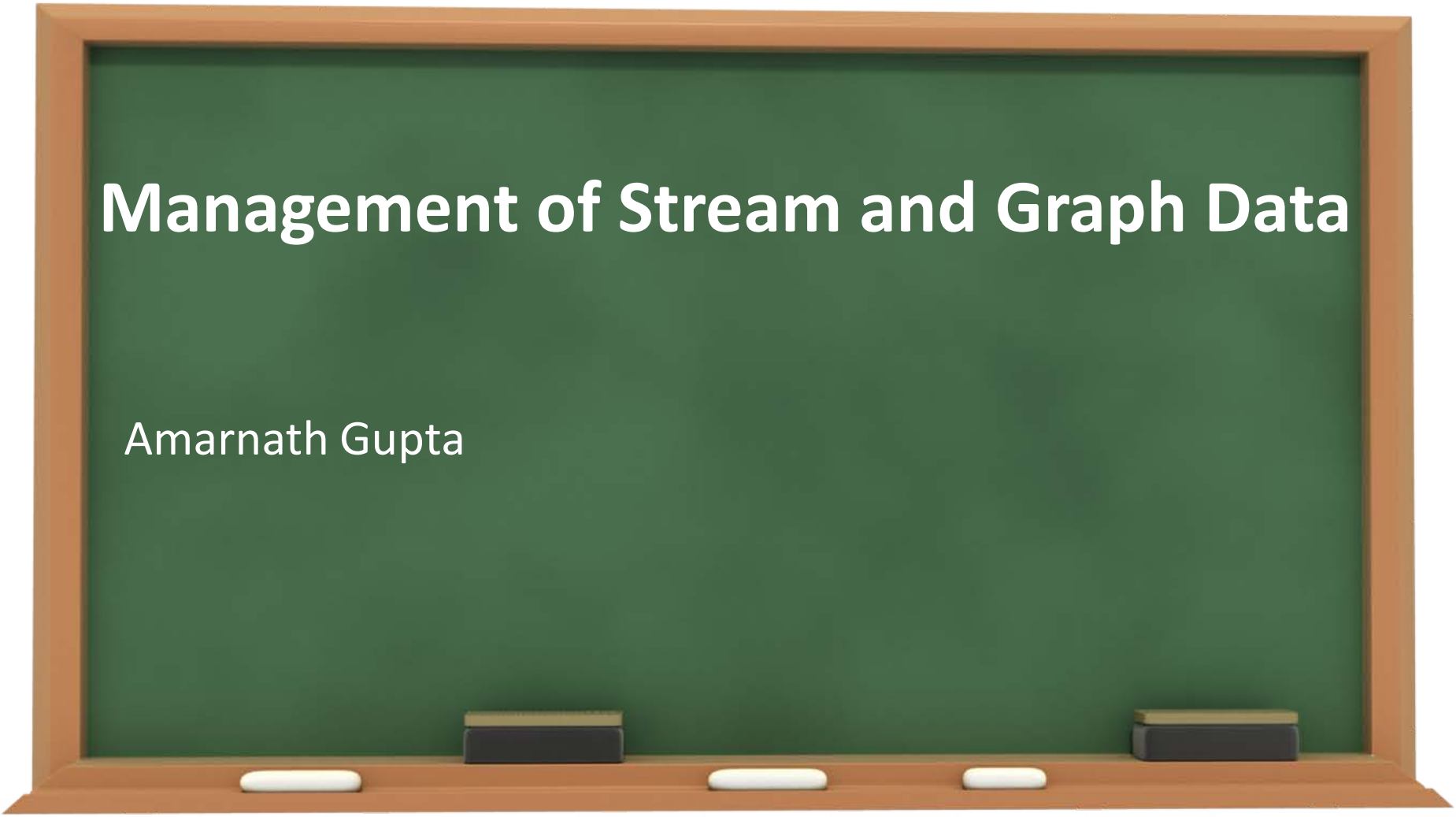
Management of Stream and Graph Data

Amarnath Gupta



Prologue

- Two trends in developing “new model” data management systems
  - How to build on top of existing data management systems
    - For example, how to represent tree-structured data (documents, XML, …) in a relational system
      - Using relational storage
      - Minimally extending the data operator set to accommodate the properties of the new model
  - How to build a “native” system that
    - Exploits the properties of the new model
    - Develops new and efficient algorithms for “natural” operators of the model
- Often, as technology matures, output of the second category is adapted/co-opted into traditional systems

Examples of New Models

- Structure-based Models
  - Multidimensional arrays
    - Started in mid-90s, now carried out under SciDB and related efforts
  - Graph data
    - Started in the 80s, now flourishing in industry, open source communities and academia
- Quality-based Models
  - Uncertain data
    - Started in the mid-80s as probabilistic relational model, now regaining importance due to quality and trust issues
- Data Property-based Models
  - Streaming data
    - Started with messaging systems, now growing in industry, open source communities and academia

STREAM DATA

The 8 Requirements of Real-Time Stream Processing

Michael Stonebraker
Computer Science and Artificial Intelligence Laboratory, M.I.T., and StreamBase Systems, Inc.
stonebraker@csail.mit.edu

Uğur Çetintemel
Department of Computer Science, Brown University, and StreamBase Systems, Inc.
ugur@cs.brown.edu

Stan Zdonik
Department of Computer Science, Brown University, and StreamBase Systems, Inc.
sbz@cs.brown.edu

ABSTRACT

Applications that require real-time processing of high-volume data steams are pushing the limits of traditional data processing infrastructures. These stream-based applications include market feed processing and electronic trading on Wall Street, network and infrastructure monitoring, fraud detection, and command and control in military environments. Furthermore, as the "sea change" caused by cheap micro-sensor technology takes hold, we expect to see everything of material significance on the planet get "sensor-tagged" and report its state or location in real time. This sensorization of the real world will lead to a "green field" of novel monitoring and control applications with high-volume and low-latency processing requirements.

Recently, several technologies have emerged—including off-the-shelf stream processing engines—specifically to address the challenges of processing high-volume, real-time data without requiring the use of custom code. At the same time, some existing software technologies, such as main memory DBMSs and rule engines, are also being "repurposed" by marketing departments to address these applications.

In this paper, we outline eight requirements that a system software should meet to excel at a variety of real-time stream processing applications. Our goal is to provide high-level guidance to information technologists so that they will know what to look for when evaluation alternative stream processing solutions. As such, this paper serves a purpose comparable to the requirements papers

Similar requirements are present in monitoring computer networks for denial of service and other kinds of security attacks. Real-time fraud detection in diverse areas from financial services networks to cell phone networks exhibits similar characteristics. In time, process control and automation of industrial facilities, ranging from oil refineries to corn flakes factories, will also move to such "firehose" data volumes and sub-second latency requirements.

There is a "sea change" arising from the advances in micro-sensor technologies. Although RFID has gotten the most press recently, there are a variety of other technologies with various price points, capabilities, and footprints (e.g., mote [1] and Lojack [2]). Over time, this sea change will cause everything of material significance to be sensor-tagged to report its location and/or state in real time.

Military has been an early driver and adopter of wireless sensor network technologies. For example, the US Army has been investigating putting vital-signs monitors on all soldiers. In addition, there is already a GPS system in many military vehicles, but it is not connected yet into a closed-loop system. Using this technology, the army would like to monitor the position of all vehicles and determine, in real time, if they are off course.

Other sensor-based monitoring applications will come over time in non-military domains. Tagging will be applied to customers at amusement parks for ride management and prevention of lost children. More sophisticated "easy-pass" systems will allow

A Slightly Modified Version of 8 Rules

- **Rule 1:** Keep the Data Moving (straight-through architecture, no-store, no-poll)
- **Rule 2:** Query using SQL on Streams (use a familiar query language)
- **Rule 3:** Handle Stream Imperfections (Delayed, Missing and Out-of-Order Data)
- **Rule 4:** Generate Predictable (deterministic) Outcomes i.e., respect order while processing
- **Rule 5:** Integrate Stored and Streaming Data
- **Rule 6:** Guarantee Data Safety and Availability (must be available with integrity at all times). Use hot backup, and real-time failover
- **Rule 7:** Partition and Scale Applications Automatically (elasticity)
- **Rule 8:** Process and Respond Instantaneously (low latency)

What makes streams different?

- In a traditional DBMS
  - Data is stored – it can be very large, but it is finite at any time
  - Queries come at random – once a query is answered, it is not persisted
- In a data stream management system (DSMS)
  - The data keeps coming continuously, i.e., the data is infinite
  - Any piece of data is available for processing for a short time
  - Queries are registered and are often “standing” (or continuous)
  - Often the results are expected to be (near) real-time
- Scientific examples
  - Data from sensor networks (including mobile applications)
  - Social network data (including participatory sensing)
  - Data from communication systems

DBMS vs. DSMS

Database management system (DBMS)	Data stream management system (DSMS)
Persistent data (relations)	volatile data streams
Random access	Sequential access
One-time queries	Continuous queries
(theoretically) unlimited secondary storage	limited main memory
Only the current state is relevant	Consideration of the order of the input
relatively low update rate	potentially extremely high update rate
Little or no time requirements	Real-time requirements
Assumes exact data	Assumes outdated/inaccurate data
Plannable query processing	Variable data arrival and data characteristics

In many applications, streaming data must be processed along with stored data

A Data Model for Streams

- A stream S is a (possibly) infinite bag (multiset) of elements <s,τ> where s is a tuple belonging to the schema of S and τ is the timestamp of the element
- Example: Traffic Data from California Dept. of Transportation http://pems.dot.ca.gov/ (data every 30 sec.)

Name	Comment	Units
Timestamp	Sample time as reported by the field element as MM/DD/YYYY HH24:MI:SS.	
Station	Unique station identifier. Use this value to cross-reference with Metadata files.	
Lane N Flow	Number of vehicle that passed over the detector during the sample period. N ranges from 1 to the number of lanes at the location.	Veh/Sample Period
Lane N Occupancy	Occupancy of the lane during the sample period expressed as a decimal number between 0 and 1. N ranges from 1 to the number of lanes at the location.	%
Lane N Speed	Speed as measured by the detector. Empty if the detector does not report speed. N ranges from 1 to the number of lanes at the location.	Mph

A Stream (Event) Processor

- The basic model
  - ordered tuples, often with an explicit timestamp
- Declaring an event/stream

Esper (in-memory processor, available from http://esper.codehaus.org/)

```
Create schema LaneFlow(laneNum int, Flow int, Occupancy int, Speed float)

Create schema trafficData as (tStamp long, stationID int, LaneFlow[])

        starttimestamp tStamp
```

StreamBase (allows stream persistence, available from http://streambase.com)

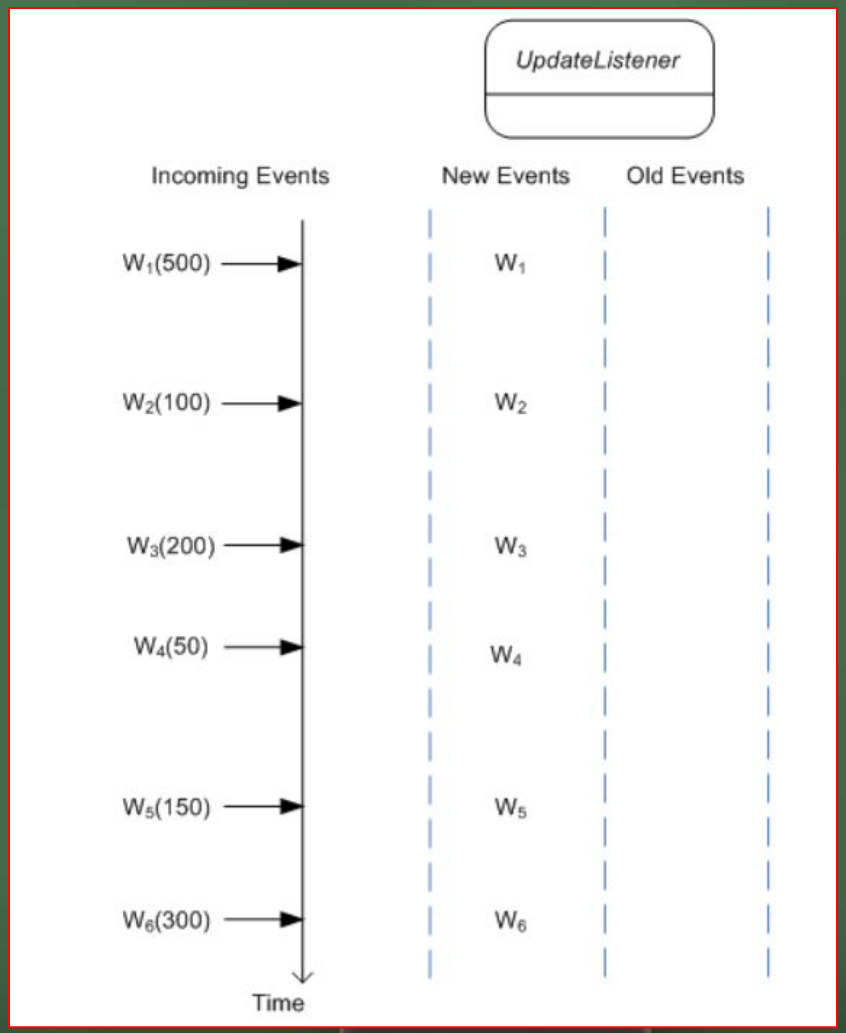
```
Create schema trafficData as (tStamp long, stationID int, laneNum int, Flow int, Occupancy int,
                              Speed float)

Create input stream trafficStream  trafficData
```

Windowing

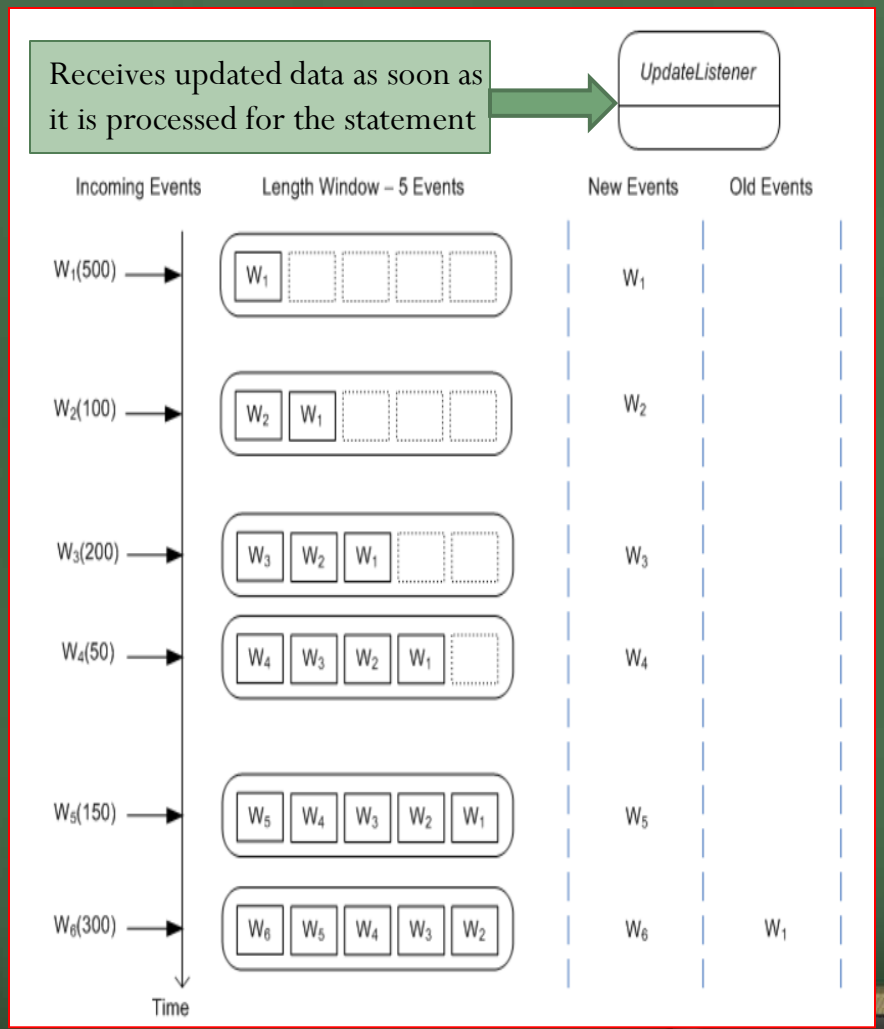
- We cannot create blocking operations on streams
  - But we want to compute joins and aggregate functions like count, average
- We create windows on streams
  - Within a window we can consider the data to be a snapshot relation and perform table-like operations on it
  - Then we get the next block of data by moving the window
- Types of window
  - How to shift
    - **Sliding**: move window on k ticks/time continuously or in blocks
    - **Tumbling**: create new window every k time-ticks or W size
  - How to construct a block
    - **By time**: window contains tuples within a certain time range, size varies with data rate
    - **By size**: at any time window contains a fixed amount of items, new data displaces old

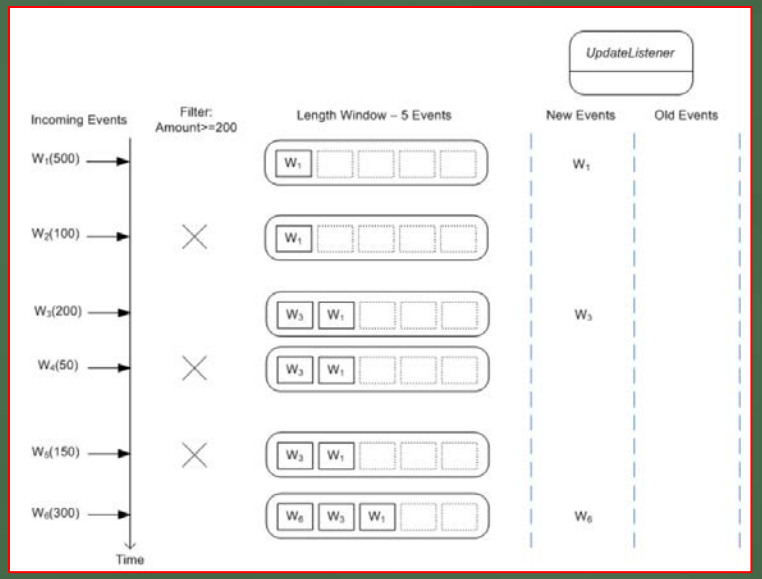
Window-based Selection (Esper)

select * from MyStream

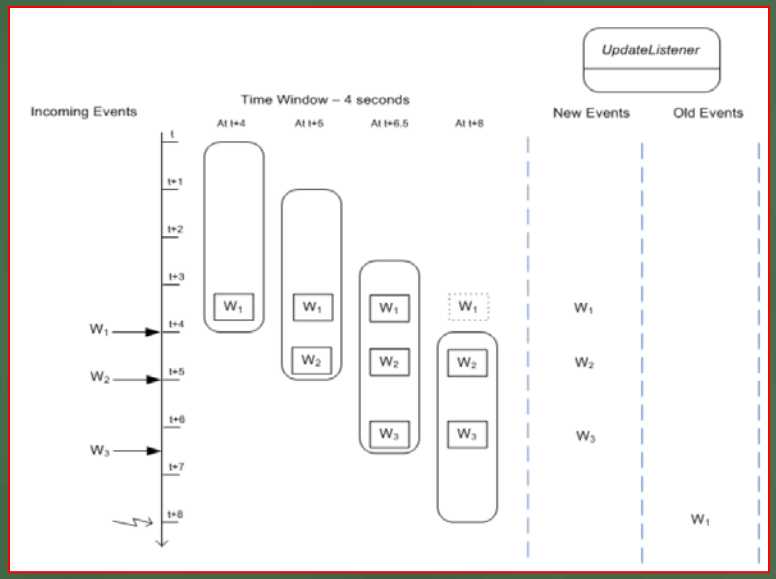


select * from MyStream.win:length(5)

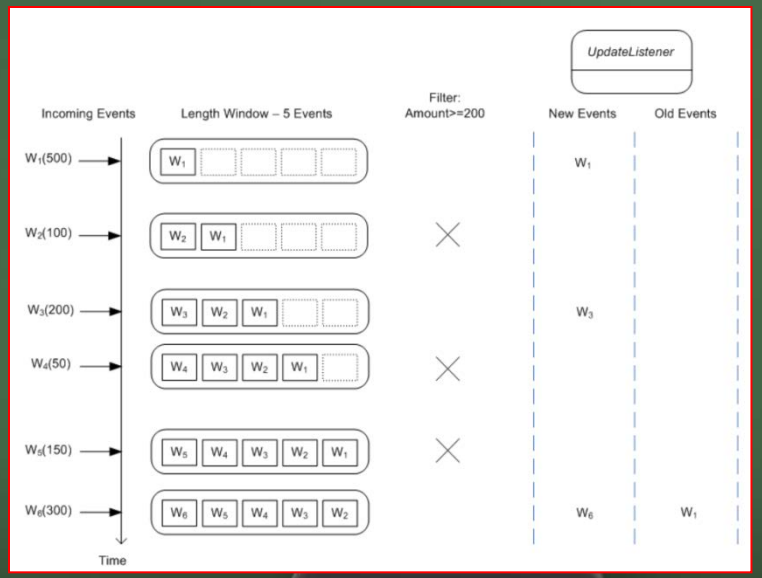




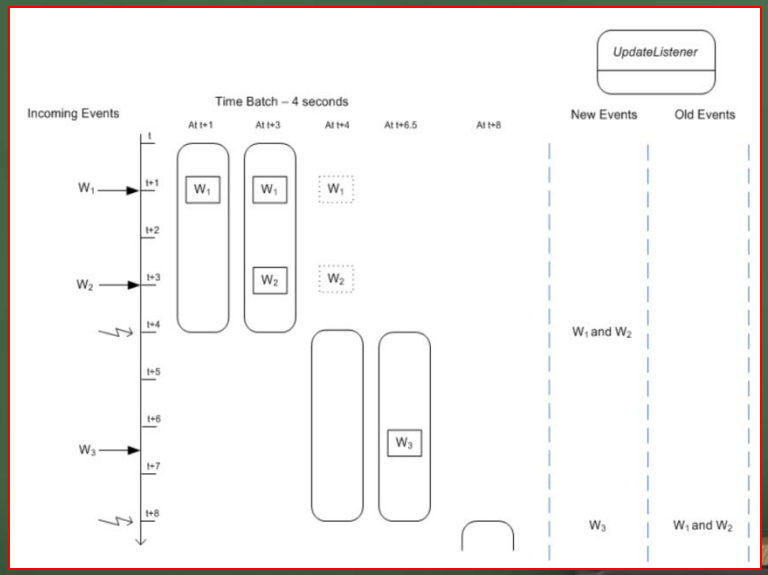
select * from MyStream(amount>=200).win:length(5)



select * from MyStream.win:time(4sec)



select * from MyStream.win:length(5) where amount >200



select * from MyStream.win:time_batch(4sec)

Output Control

select irstream value from MyStream.win:time(5.5 sec)

Time	Value	Input Stream	Remove Stream
0.2	W1	W1	
0.8	W2	W2	
1.0			
1.2			
1.5	W3, W4	W3, W4	
2.0			
2.1	W5	W5	
2.2			
2.5			
3.0			
3.2			
3.5	W6	W6	
4.0			
4.3	W7	W7	
4.9	W8	W8	
5.0			
5.2			
5.7			W1
5.9	W9	W9	
6.0			
6.2			
6.3			W2
7.0			W3, W4
7.2			

Output Control

select irstream value from MyStream.win:time(5.5 sec) **output every 1 seconds**

Time	Value	Input Stream	Remove Stream
0.2	W1		
0.8	W2		
1.0			
1.2		W1, W2	
1.5	W3, W4		
2.0			
2.1	W5		
2.2		W3, W4, W5	
2.5			
3.0			
3.2		null	
3.5	W6		
4.0			
4.2		W6	
4.3	W7		
4.9	W8		
5.0			
5.2		W7, W8	
5.7			
5.9	W9		
6.0			
6.2		W9	W1
6.3			
7.0			
7.2		Null	W2, W3, W4

Output Control

select irstream value from MyStream.win:time(5.5 sec) output **last** every 1 seconds

Time	Value	Input Stream	Remove Stream
0.2	W1		
0.8	W2		
1.0			
1.2		W2	
1.5	W3, W4		
2.0			
2.1	W5		
2.2		W5	
2.5			
3.0			
3.2		null	
3.5	W6		
4.0			
4.2		W6	
4.3	W7		
4.9	W8		
5.0			
5.2		W8	
5.7			
5.9	W9		
6.0			
6.2		W9	W1
6.3			
7.0			
7.2			W4

Output Control

select irstream value from MyStream.win:time(5.5 sec) output **snapshot** every 1 seconds

Time	Value	Input Stream	Remove Stream
0.2	W1		
0.8	W2		
1.0			
1.2		W1, W2	
1.5	W3, W4		
2.0			
2.1	W5		
2.2		W1, W2, W3, W4, W5	
2.5			
3.0			
3.2		W1, W2, W3, W4, W5	
3.5	W6		
4.0			
4.2		W1, W2, W3, W4, W5, W6	
4.3	W7		
4.9	W8		
5.0			
5.2		W1, W2, W3, W4, W5, W6, W7, W8	
5.7			
5.9	W9		
6.0			
6.2		W2, W3, W4, W5, W6, W7, W8, W9	
6.3			
7.0			
7.2		W5, W6, W7, W8, W9	

Window-Based Stream Partitioning

StreamBase

```
CREATE OUTPUT STREAM TrafficStats AS
SELECT openval() AS StartOfTimeSlice,
avg(Occupancy) AS AvgCarsPerSecond,
stdev(Occupancy) AS StdevCarsPerSecond,
lastval(Occupancy) AS LastCarsPerSecond,
StationNum
FROM trafficStream [SIZE 20 ADVANCE 1 ON
        StartOfTimeSlice PARTITION BY
         StationNum]
GROUP BY StationNum;
```

Esper

```
CREATE CONTEXT  TrafficPerStation
      PARTITION BY StationNum from
trafficStream

    CONTEXT TrafficPerStation
    SELECT timestamp,
       avg(Occupancy) AS AvgCarsPerSecond,
       stdev(Occupancy) AS StdevCarsPerSecond,
       lastval(Occupancy) AS LastCarsPerSecond,
    FROM trafficStream.win:length(20)
```

Pattern Specification

Streambase

```
SELECT A.id AS fi, C.id AS fo
FROM PATTERN A → !B → C WITHIN 5 TIME
WHERE B.id == A.id
INTO out;
```

Esper

```
select a.custId, sum(a.price + b.price)
from pattern
        [every a=ServiceOrder → b=ProductOrder(custId = a.custId)
                where timer:within(1 min)].win:time(2 hour)
where a.name in ('Repair', b.name)
group by a.custId
having sum(a.price + b.price) > 100
```

Toward a Distributed DSMS for Large, High Velocity Data

- Goals
  - Guaranteed data processing
  - Fault tolerance
  - Horizontal scalability
  - Allows one to use a high-level programming language

What Is Apache Hadoop?

The Apache™ Hadoop® project develops open-source software for reliable, scalable, distributed computing.

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

The project includes these modules:

- **Hadoop Common**: The common utilities that support the other Hadoop modules.
- **Hadoop Distributed File System (HDFS™)**: A distributed file system that provides high-throughput access to application data.
- **Hadoop YARN**: A framework for job scheduling and cluster resource management.
- **Hadoop MapReduce**: A YARN-based system for parallel processing of large data sets.

Two Recent Distributed Stream Platforms

S4 distributed stream computing platform

home doc code API get involved team download

What is S4?

S4 is a general-purpose,near real-time, distributed, decentralized, scalable, event-driv
applications for processing continuous unbounded streams of data.

S4 0.5 focused on providing a functional complete refactoring.

S4 0.6 builds on this basis and brings plenty of exciting features, in particular:

- **major performance improvements**: stream throughput improved by 1000 % (2
- major configurability and usability improvements, for both the S4 platform and d

What are the cool features?

Flexible deployment:

- Application packages are standard jar files (suffixed `.s4r`)
- Platform modules for customizing the platform are standard jar files
- By default keys are homogeneously sparsed over the cluster: helps balance the lo

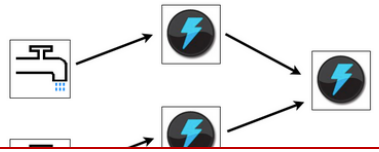
Storm

Distributed and fault-tolerant realtime computation

about documentation blog downloads community

Storm is a free and open source distributed realtime computation system. Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing. Storm is simple, can be used with any programming language, and is a lot of fun to use!

Storm has many use cases: realtime analytics, online machine learning, continuous computation, distributed RPC, ETL, and more. Storm is fast: a benchmark clocked it at over **a million tuples processed per second per node**. It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate.



Download Storm

Source code

Star 6,508

Community

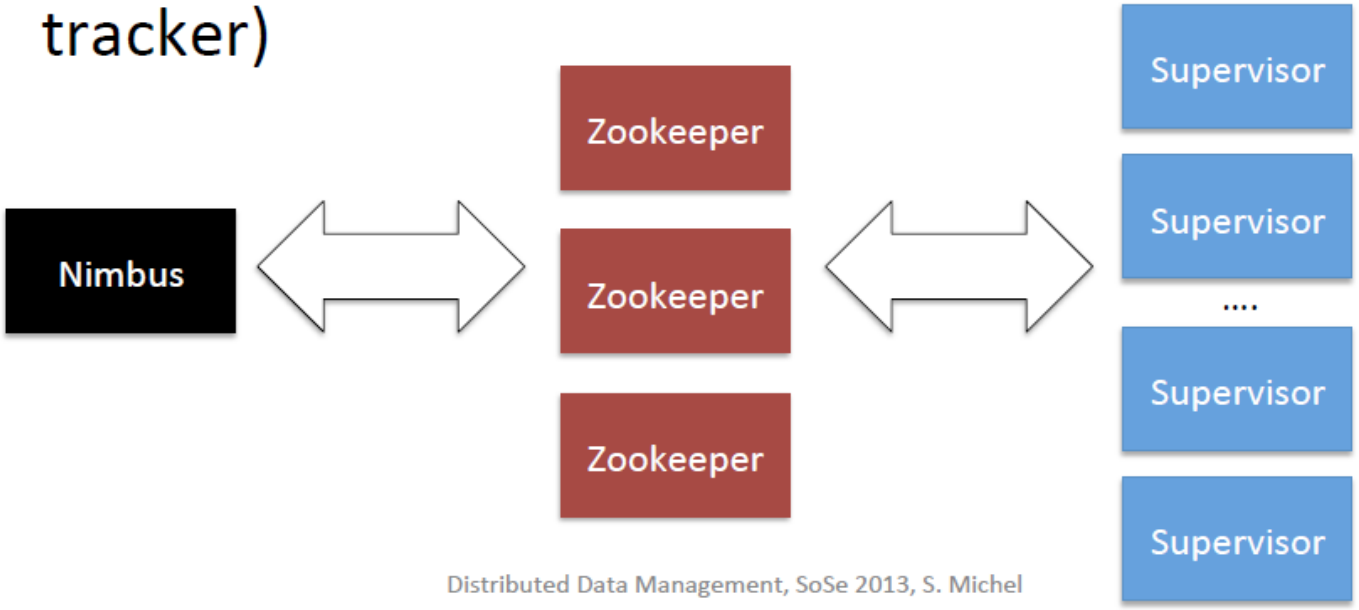
Follow @stormprocessor

Mailing list

Trident is a high-level abstraction for doing realtime computing on top of Storm. It allows you to seamlessly intermix high throughput (millions of messages per second), stateful stream processing with low latency distributed querying. If you're familiar with high level batch processing tools like Pig or Cascading, the concepts of Trident will be very familiar – Trident has joins, aggregations, grouping, functions, and filters. In addition to these, Trident adds primitives for doing stateful, incremental processing on top of any database or persistence store. Trident has consistent, exactly-once semantics, so it is easy to reason about Trident topologies.

Storm Cluster Setup

- Using Apache Zookeeper for coordination
- Supervisor: worker nodes (like Hadoop task tracker)
- Nimbus: coordinator node (like Hadoop job tracker)



Apache ZooKeeper™

Welcome to Apache ZooKeeper™

Apache ZooKeeper is an effort to develop and maintain an open-source server which enables highly reliable distributed coordination.

What is ZooKeeper?

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services, applications initially usually skimp on them ,which make them brittle in the presence of change and difficult to manage. Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed.

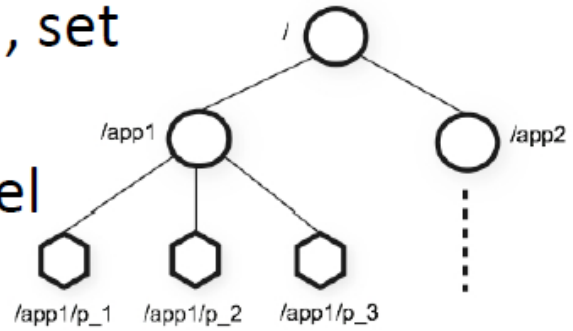
Learn more about ZooKeeper on the **ZooKeeper Wiki**.

Getting Started

Start by installing ZooKeeper on a single machine or a very small cluster.

1. **Learn about** ZooKeeper by reading the documentation.
2. **Download** ZooKeeper from the release page.

- Hierarchical data model, simple API: create, delete, exists, get data, set data, get children, sync
- Used to implement higher level applications



http://zookeeper.apache.org/doc/trunk/zookeeperOver.html

Zookeeper Guarantees

- **Sequential Consistency:** Updates from a client will be applied in the order that they were sent.
- **Atomicity:** Updates either succeed or fail.
- **Single System Image:** A client will see the same view of the service regardless of the server that it connects to.
- **Reliability:** Once an update has been applied, it will persist from that time forward until a client overwrites the update.
- **Timeliness:** The clients view of the system is guaranteed to be up-to-date within a certain time bound.

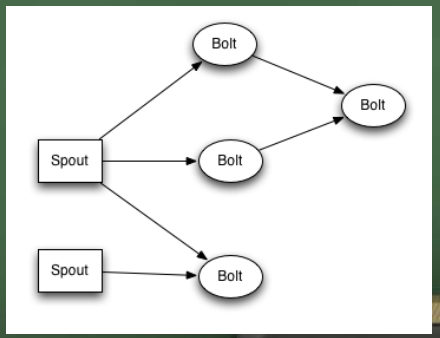
Storm and Zookeeper

- Storm uses Zookeeper for
  - Discovery of nodes
  - Storing the state of Nimbus and Supervisor processes
  - Guaranteed message processing and tracking
  - Storing statistics
- The actual heavy lifting (i.e., internode communication) uses a library called zero MQ

```
import zmq
import time
context = zmq.Context()

subscriber = context.socket (zmq.SUB)
subscriber.connect ("tcp://192.168.55.112:5556")
subscriber.connect ("tcp://192.168.55.201:7721")
subscriber.setsockopt (zmq.SUBSCRIBE, "NASDAQ")

publisher = context.socket (zmq.PUB)
publisher.bind ("ipc://nasdaq-feed")

while True:
    message = subscriber.recv()
    publisher.send (message)
```

The Storm Computing Model

- **Spouts:** A spout is a source of streams. For example, a spout may read tuples off of a Kestrel queue and emit them as a stream. Or a spout may connect to the Twitter API and emit a stream of tweets.
- **Bolts:** A bolt consumes any number of input streams, does some processing, and possibly emits new streams. Complex stream transformations, require multiple steps and thus multiple bolts. Bolts can do anything from run functions, filter tuples, do streaming aggregations, do streaming joins, talk to databases, and more.
- **Topology:** A topology is a graph of stream transformations where each node is a spout or bolt. Edges in the graph indicate which bolts are subscribing to which streams. When a spout or bolt emits a tuple to a stream, it sends the tuple to every bolt that subscribed to that stream.



A Simple Non-trivial Example

Credit: Svend Vanderveken

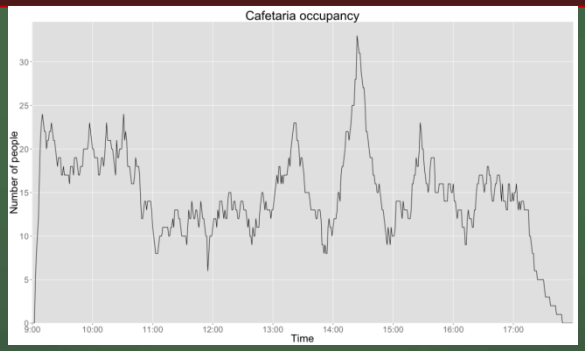
- Scenario
  - There is a building with a number of rooms. There are a bunch of people wearing sensors going into and coming out of rooms. Every time some one enters and leaves, the sensors emit data giving out that information with a timestamp.
- Goal: create an occupancy timeline for each room.
- Data schema: (eventType, userID, timeStamp, roomID, dataID, corrID)
  - Events are not guaranteed to respect chronological order

Input and Output

- What comes in

{"eventType": "ENTER", "userId": "John_5", "time": 1374922058918, "roomId": "Cafetaria", "id": "bf499c0bd09856e7e0f68271336103e0A", "corrId": "bf499c0bd09856e7e0f68271336103e0"}
{"eventType": "ENTER", "userId": "Zoe_15", "time": 1374915978294, "roomId": "Conf1", "id": "3051649a933a5ca5aeff0d951aa44994A", "corrId": "3051649a933a5ca5aeff0d951aa44994"}
{"eventType": "LEAVE", "userId": "Jenny_6", "time": 1374934783522, "roomId": "Conf1", "id": "6abb451d45061968d9ca01b984445ee8B", "corrId": "6abb451d45061968d9ca01b984445ee8"}
{"eventType": "ENTER", "userId": "Zoe_12", "time": 1374921990623, "roomId": "Hall", "id": "86a691490fff3fd4d805dce39f832b31A", "corrId": "86a691490fff3fd4d805dce39f832b31"}
{"eventType": "LEAVE", "userId": "Marie_11", "time": 1374927215277, "roomId": "Conf1", "id": "837e05916349b42bc4c5f65c0b2bca9dB", "corrId": "837e05916349b42bc4c5f65c0b2bca9d"}
{"eventType": "ENTER", "userId": "Robert_8", "time": 1374911746598, "roomId": "Annex1", "id": "c461a50e236cb5b4d6b2f45d1de5cbb5A", "corrId": "c461a50e236cb5b4d6b2f45d1de5cbb5"}

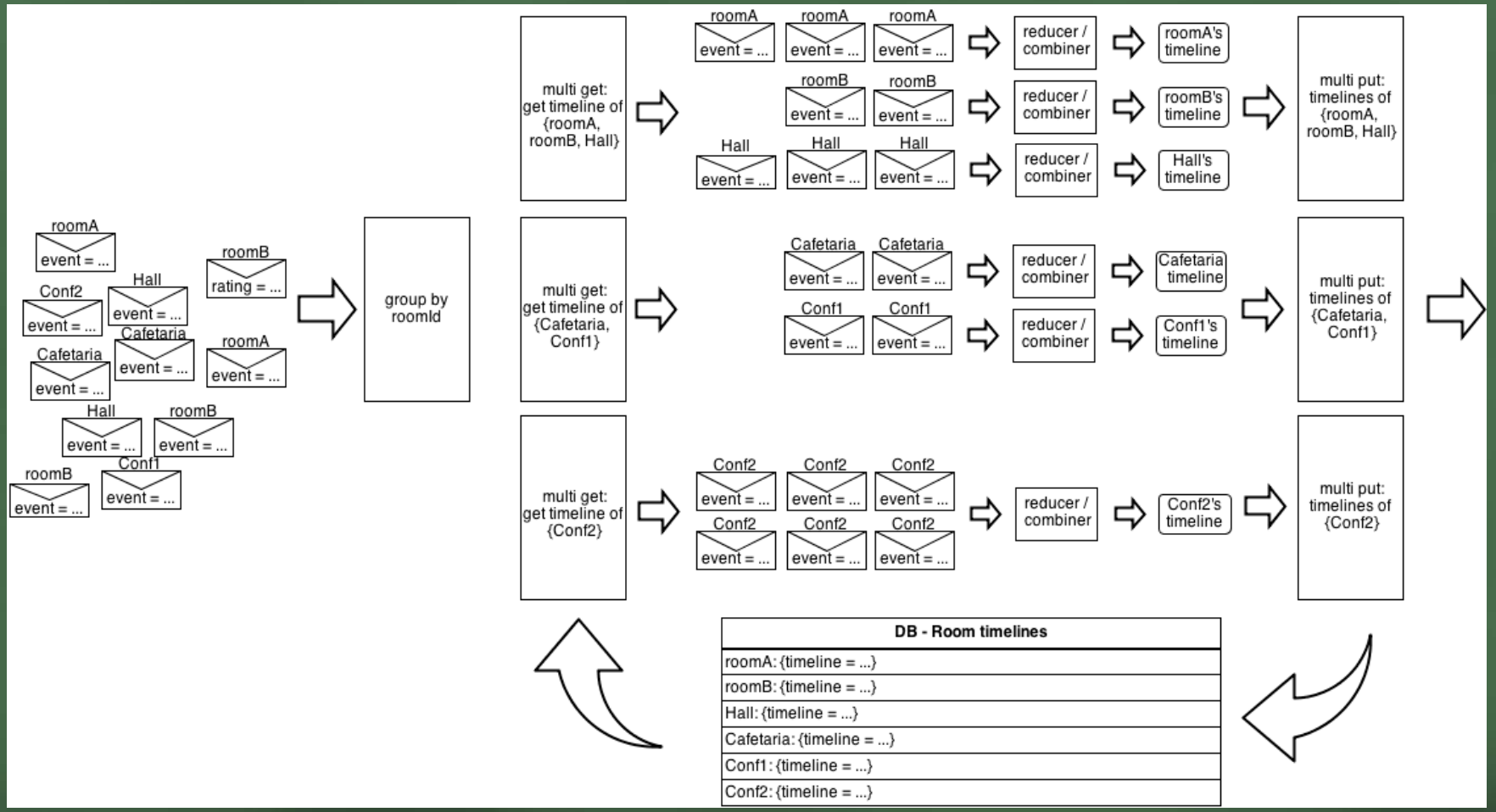
- What the system should produce



An intermediate output

{"roomId":"Cafetaria","sliceStartMillis":1374926400000,"occupancies":[11,12,12,12,13,15,15,14,1

The Computation Scheme



Trident Topology

Goal: build a minute-by-minute occupancy timeline of each room

- Read input events in JSON

```
TridentTopology topology = new TridentTopology();

topology
.newStream("occupancy", new SimpleFileStringSpout("data/events.json", "rawOccupancyEvent"))
.each(new Fields("rawOccupancyEvent"), new EventBuilder(), new Fields("occupancyEvent"))
```

- Gather "enter" and "leave" events into "presence periods"

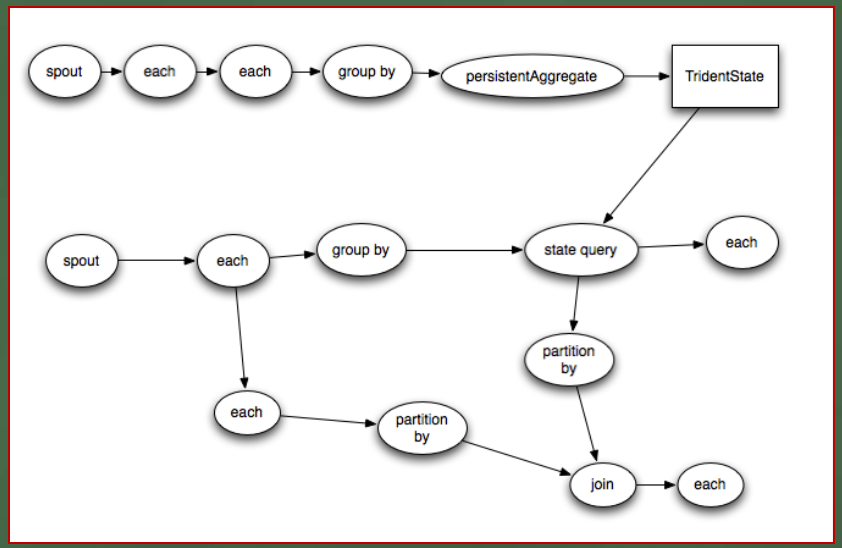
```
.each(new Fields("occupancyEvent"), new ExtractCorrelationId(), new Fields("correlationId"))
.groupBy(new Fields("correlationId"))
.persistentAggregate( PeriodBackingMap.FACTORY, new Fields("occupancyEvent"), new PeriodBuilder(),
new Fields("presencePeriod"))
.newValuesStream()
```

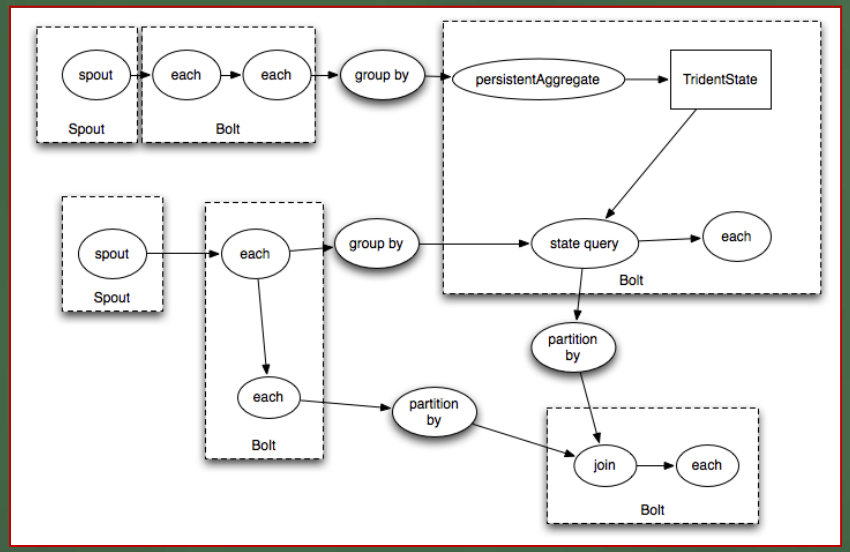
- Build room timelines

```
.each(new Fields("presencePeriod"), new IsPeriodComplete())
.each(new Fields("presencePeriod"), new BuildHourlyUpdateInfo(), new Fields("roomId", "roundStartTi
.groupBy(new Fields("roomId", "roundStartTime"))
.persistentAggregate( TimelineBackingMap.FACTORY, new Fields("presencePeriod","roomId", "roundSta
 new TimelineUpdater(), new Fields("hourlyTimeline"));
```

Trident Topologies

- Trident topologies compile down into as efficient of a Storm topology as possible. Tuples are only sent over the network when a repartitioning of the data is required, such as if you do a groupBy





Trident Topology

Goal: build a minute-by-minute occupancy timeline of each room

- Read input events in JSON

```
TridentTopology topology = new TridentTopology();

topology
.newStream("occupancy", new SimpleFileStringSpout("data/events.json", "rawOccupancyEvent"))
.each(new Fields("rawOccupancyEvent"), new EventBuilder(), new Fields("occupancyEvent"))
```

- Gather "enter" and "leave" events into "presence periods"

```
.each(new Fields("occupancyEvent"), new ExtractCorrelationId(), new Fields("correlationId"))
.groupBy(new Fields("correlationId"))
.persistentAggregate( PeriodBackingMap.FACTORY, new Fields("occupancyEvent"), new PeriodBuilder(),
new Fields("presencePeriod"))
.newValuesStream()
```

- Build room timelines

```
.each(new Fields("presencePeriod"), new IsPeriodComplete())
.each(new Fields("presencePeriod"), new BuildHourlyUpdateInfo(), new Fields("roomId", "roundStartTi
.groupBy(new Fields("roomId", "roundStartTime"))
.persistentAggregate( TimelineBackingMap.FACTORY, new Fields("presencePeriod","roomId", "roundSta
 new TimelineUpdater(), new Fields("hourlyTimeline"));
```

Trident Abstractions

- Fields and Tuples
  - Suppose there is a stream stream(x, y, z)
    - stream.each(new Fields("y"), new MyFilter())
    - public class MyFilter extends BaseFilter {
      - public boolean isKeep(TridentTuple tuple)
      - { return tuple.getInteger(0) < 10; }
    - }
    - stream.each(new Fields("x", "y"), new AddAndMultiply(), new Fields("added", "multiplied"));
    - public class AddAndMultiply extends BaseFunction {
      - public void execute(TridentTuple tuple, TridentCollector collector)
      - { int i1 = tuple.getInteger(0);
      - int i2 = tuple.getInteger(1);
      - collector.emit(new Values(i1 + i2, i1 * i2)); }
    - }

Tuples are internally processed in batches

Trident Topology

Goal: build a minute-by-minute occupancy timeline of each room

- Read input events in JSON

```
TridentTopology topology = new TridentTopology();

topology
.newStream("occupancy", new SimpleFileStringSpout("data/events.json", "rawOccupancyEvent"))
.each(new Fields("rawOccupancyEvent"), new EventBuilder(), new Fields("occupancyEvent"))
```

- Gather "enter" and "leave" events into "presence periods"

```
.each(new Fields("occupancyEvent"), new ExtractCorrelationId(), new Fields("correlationId"))
.groupBy(new Fields("correlationId"))
.persistentAggregate( PeriodBackingMap.FACTORY, new Fields("occupancyEvent"), new
PeriodBuilder(), new Fields("presencePeriod"))
.newValuesStream()
```

Build room timelines

```
.each(new Fields("presencePeriod"), new IsPeriodComplete())
.each(new Fields("presencePeriod"), new BuildHourlyUpdateInfo(), new Fields("roomId",
"roundStartTime"))
.groupBy(new Fields("roomId", "roundStartTime"))
.persistentAggregate( TimelineBackingMap.FACTORY, new Fields("presencePeriod","roomId",
"roundStartTime"),
  new TimelineUpdater(), new Fields("hourlyTimeline"));
```

Computing Aggregates

- Suppose we have a stream with fields *val1* and *val2*
  - stream.aggregate(new Fields("val2"), new Sum(), new Fields("sum"))
    - The output stream would only contain a single tuple with a single field called "sum", representing the sum of all "val2" fields in that batch.
  - stream.groupBy(new Fields("val1")) .aggregate(new Fields("val2"), new Sum(), new Fields("sum"))
    - the output will contain the grouping fields followed by the fields emitted by the aggregator
    - the output will contain the fields "val1" and "sum"

Trident States

- State: content of the data at any instant
  - Sometimes we want to do state updates (e.g., an external databases) so that it's like each message was only processed only once
  - Trident solves this problem by doing two things:
    - Each batch is given a unique id called the "transaction id". If a batch is retried it will have the exact same transaction id.
    - State updates are ordered among batches. That is, the state updates for batch 3 won't be applied until the state updates for batch 2 have succeeded.

Trident PersistentAggregates

- persistentAggregate is an additional abstraction that
  - takes a Trident aggregator
  - uses it to apply updates to the source of state
  - The programmer implements the "MapState" interface
    - The grouping fields will be the keys in the state, and the aggregation result will be the values in the state.
    - public interface MapState<T> extends State{
      - List<T>**multiGet**(List<List<Object>> keys);
      - List<T>**multiUpdate**(List<List<Object>> keys, List<ValueUpdater> updaters);
      - void **multiPut**(List<List<Object>> keys, List<T> vals); }

An Example

```
public class PeriodBackingMap implements IBackingMap<RoomPresencePeriod> {

        public static StateFactory FACTORY = new StateFactory() {
                public State makeState(Map conf, IMetricsContext metrics, int partitionIndex, int numPartitions) {
                        // our logic is fully idempotent => no Opaque map nor Transactional map required here...
                        return NonTransactionalMap.build(new PeriodBackingMap());
                }
        };

        public List<RoomPresencePeriod> multiGet(List<List<Object>> keys) {
                return CassandraDB.DB.getPresencePeriods(toCorrelationIdList(keys));
        }

        public void multiPut(List<List<Object>> keys, List<RoomPresencePeriod> newOrUpdatedPeriods) {
                CassandraDB.DB.upsertPeriods(newOrUpdatedPeriods);
        }

        private List<String> toCorrelationIdList(List<List<Object>> keys) {
                List<String> structuredKeys = new LinkedList();
                for (List<Object> key : keys) {
                        structuredKeys.add((String) key.get(0));
                }
                return structuredKeys;
        }

}
```

Trident Topology

Goal: build a minute-by-minute occupancy timeline of each room

- Read input events in JSON

```
TridentTopology topology = new TridentTopology();

topology
.newStream("occupancy", new SimpleFileStringSpout("data/events.json", "rawOccupancyEvent"))
.each(new Fields("rawOccupancyEvent"), new EventBuilder(), new Fields("occupancyEvent"))
```

- Gather "enter" and "leave" events into "presence periods"

```
.each(new Fields("occupancyEvent"), new ExtractCorrelationId(), new Fields("correlationId"))
.groupBy(new Fields("correlationId"))
.persistentAggregate( PeriodBackingMap.FACTORY, new Fields("occupancyEvent"), new PeriodBuilder(),
new Fields("presencePeriod"))
.newValuesStream()
```

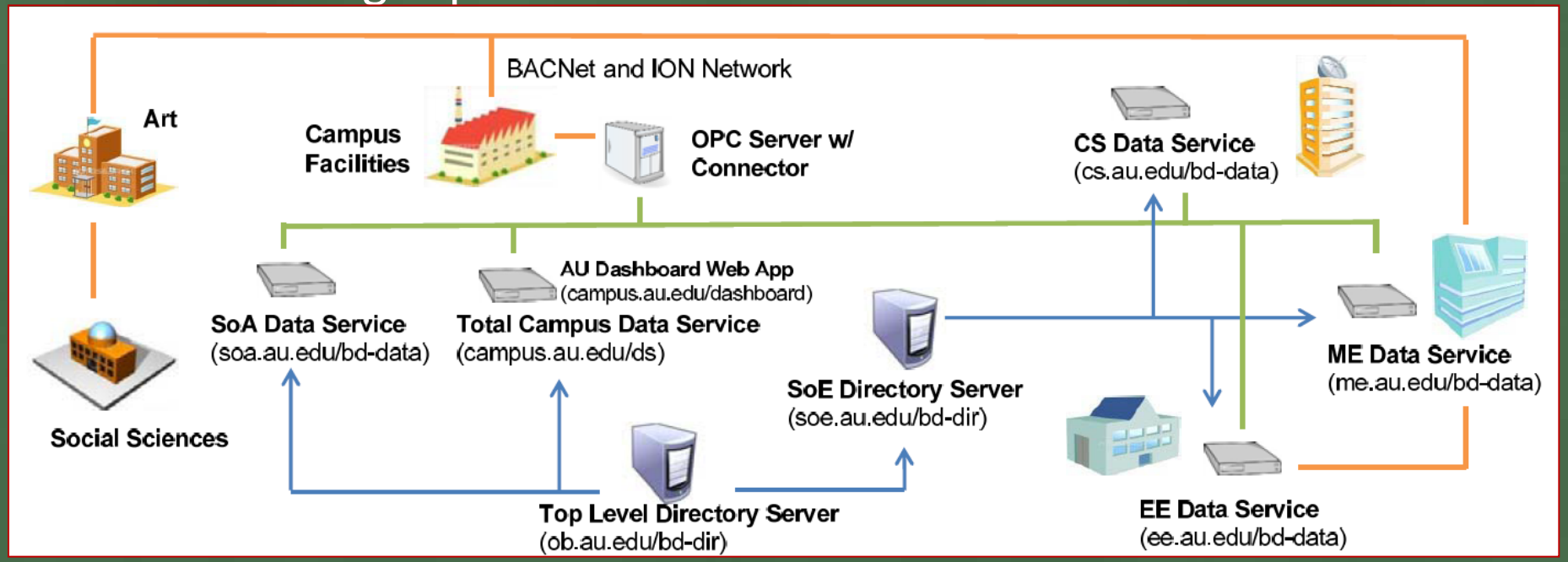
Build room timelines

```
.each(new Fields("presencePeriod"), new IsPeriodComplete())
.each(new Fields("presencePeriod"), new BuildHourlyUpdateInfo(), new Fields("roomId",
"roundStartTime"))
.groupBy(new Fields("roomId", "roundStartTime"))
.persistentAggregate( TimelineBackingMap.FACTORY, new Fields("presencePeriod","roomId",
"roundStartTime"),
 new TimelineUpdater(), new Fields("hourlyTimeline"));
```

Monitoring Energy Use

Courtesy: Yuvraj Agarwal

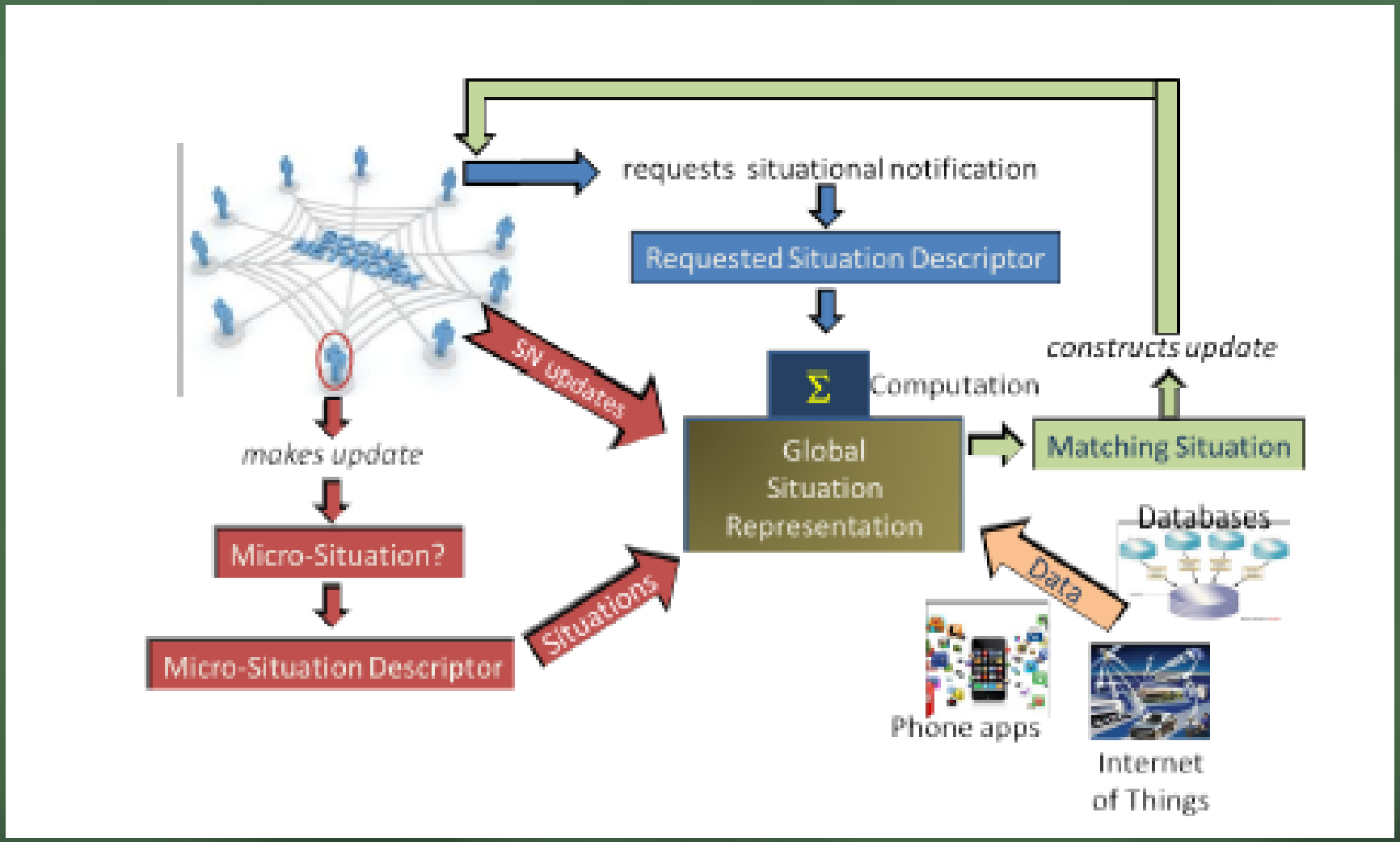
- At SDSC, we are evaluating this framework for an application called BuildingDepot



Monitoring and Control of Energy Use in Buildings

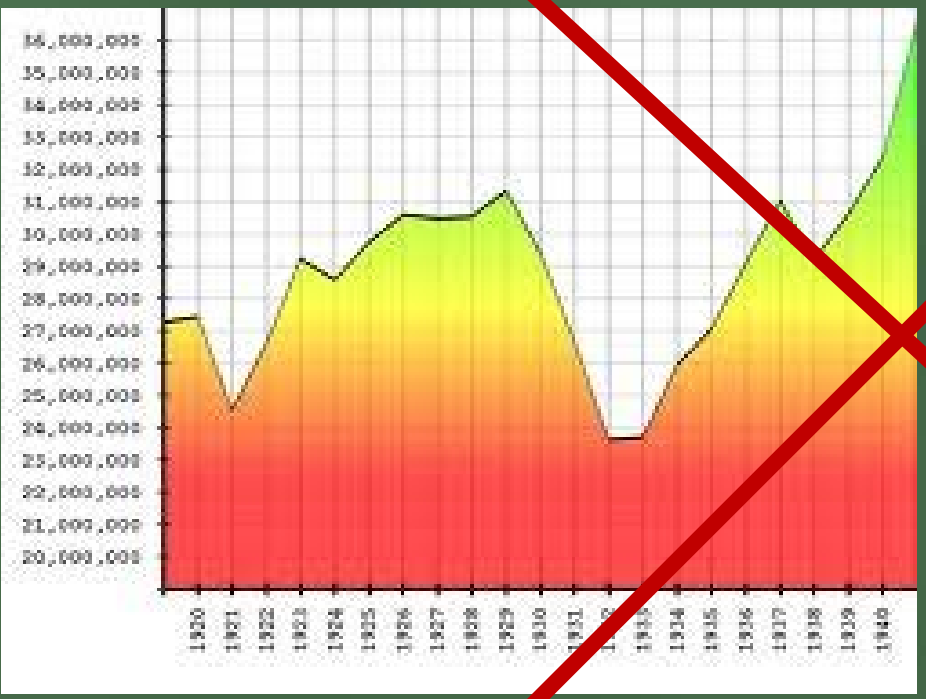
- Instrumenting more buildings with newer sensors and actuators
- 80k sensor streams now, will increase to 500k soon
- Using more spatial knowledge for effective predictive analysis

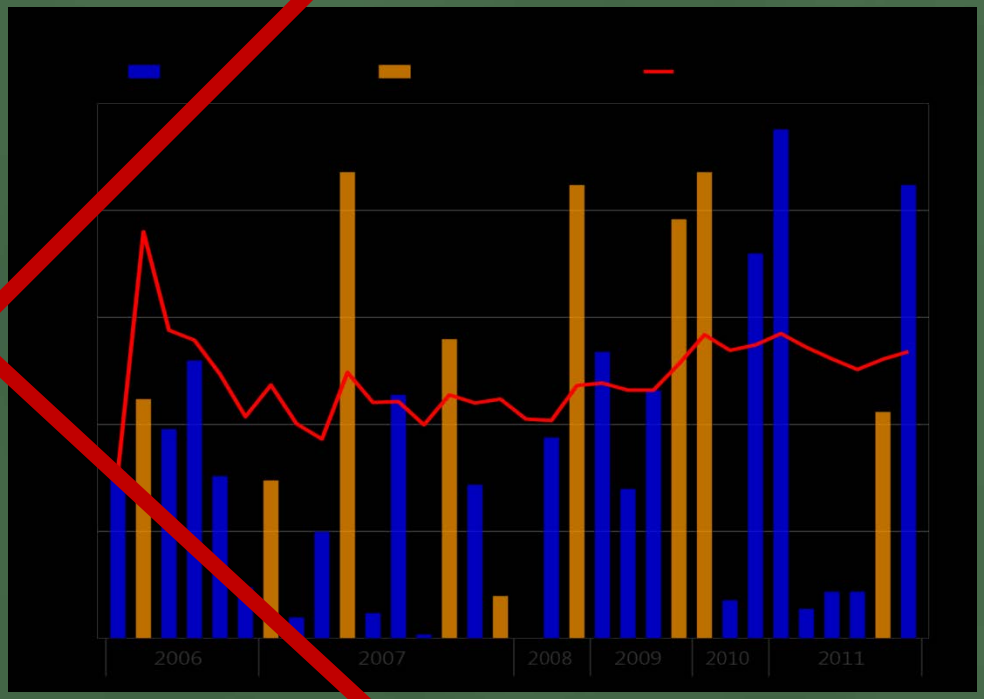
Social Life Networks



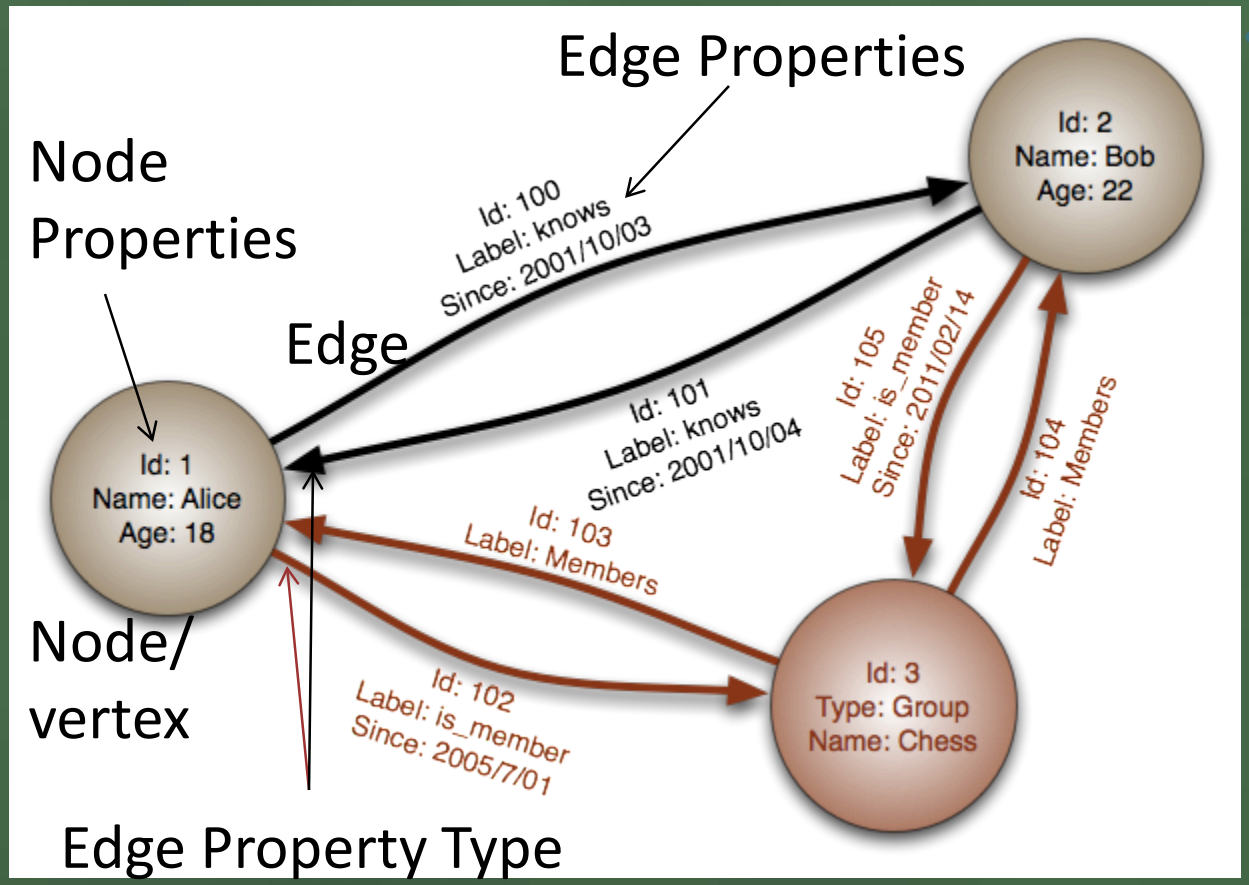
GRAPH DATA

What Do You Mean “Graphs”?





Oh! You mean Networks!!



- In this graph
  - Edges are directional
  - There are no edge weights
  - Nodes do not have their own types
  - There are no self loops
  - No logical constraints

A Real-World “Business” Problem

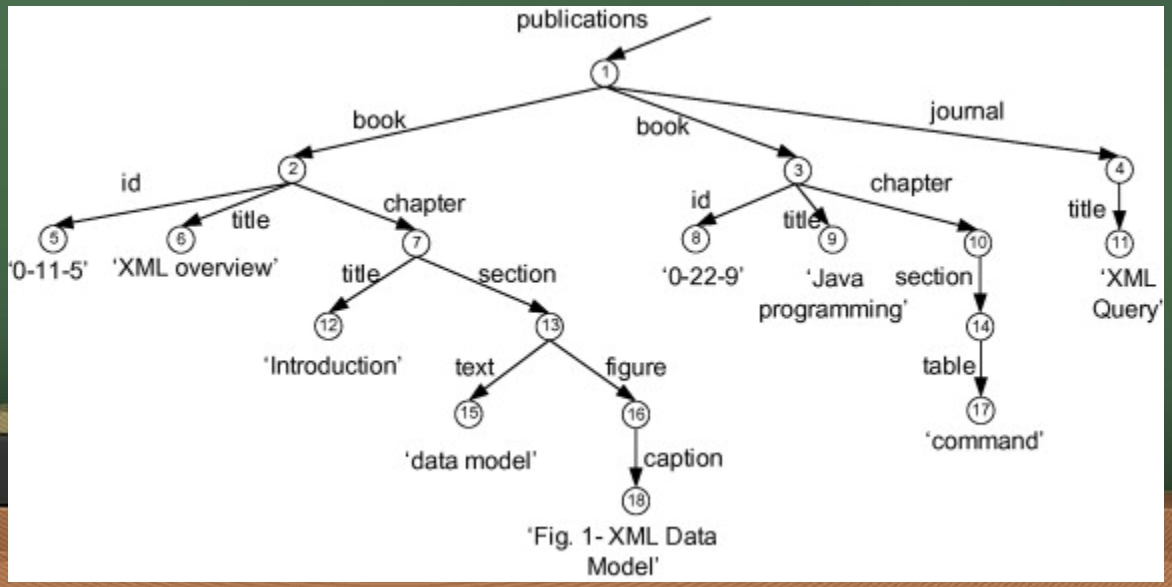
- Who should be on my board of advisors?
  - I already have A, B, C and D and need two more people who should
    - Have name recognition in their fields, which should be “around” Computer Science
    - Should have reasonably high visibility
    - Should be known to at least two of my current members
    - Should get along with C
    - Have a lot of “business connections”
    - Be independently wealthy, and if possible, an entrepreneur
    - Not be involved in any recent negative press

A Real-World Science Problem

- What are potential pharmaceutical compounds C which are potentially useful for orphan disease D?
  - These compounds should satisfy the following properties
    - They are not yet applied to D
    - They are not in the FDA approval pipeline for D
    - They have been applied to humans and model organisms
    - They operate on genes products in pathways that are relevant
      - Because these pathways are related to some phenotypes exhibited by D in humans or model organisms
    - They have not been identified as toxins for any target related to human health

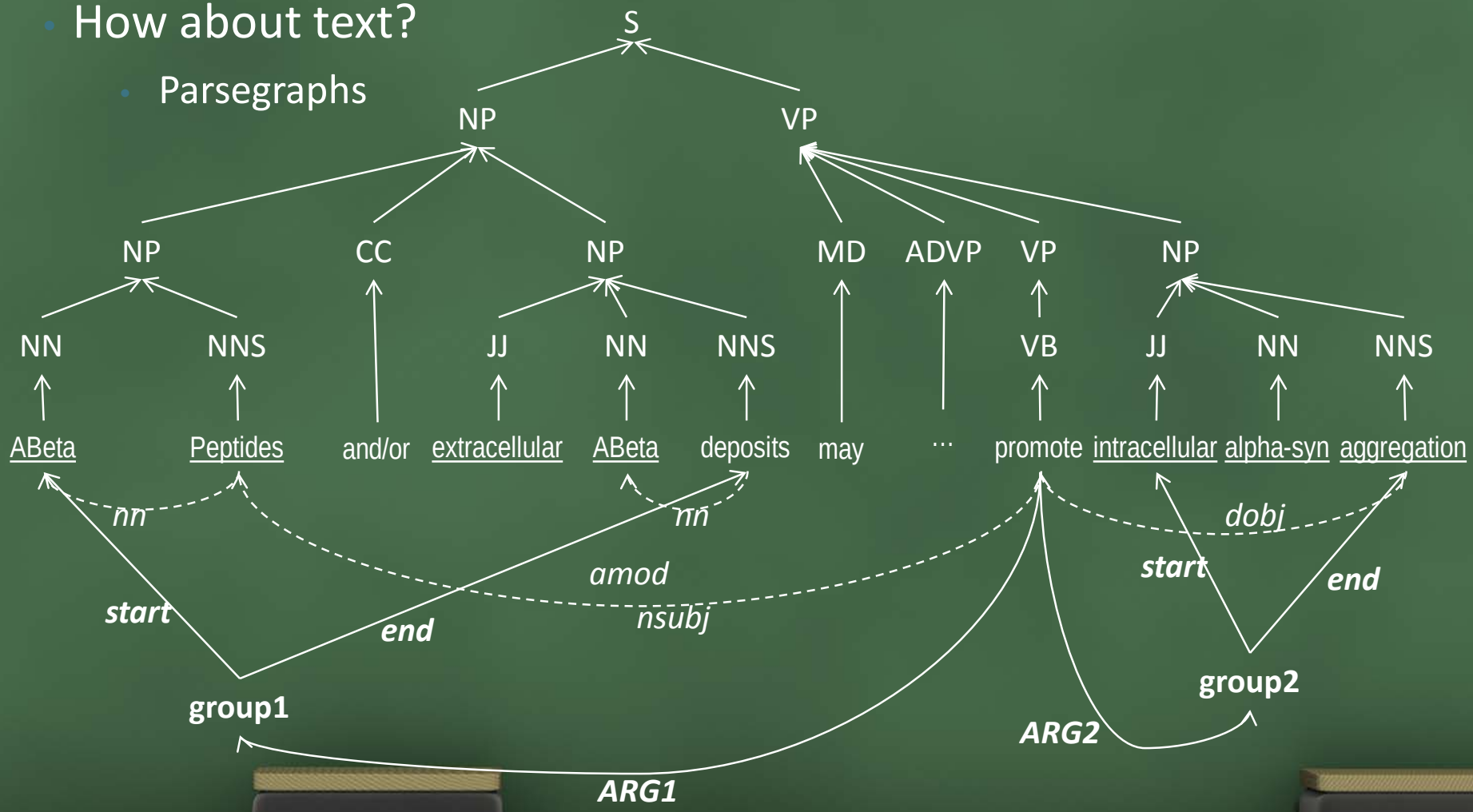
Why is a Graph a "Natural Model" for these problems?

- Structurally, most data models can be viewed as graphs
  - Does not mean they should be
  - Relation R(A,B,C), pk(A) – with tuple r1(1,2,3) can become
    - R—attrib→A, R—attrib→B, R—attrib→C, R—pk→A
    - r1—instanceOf→R
    - r 1–A→1, r1—B→2, r1—C→3
  - XML (without idRef) is modeled as an edge-labeled tree, therefore it is a graph



Why is a Graph a "Natural Model" for these problems?

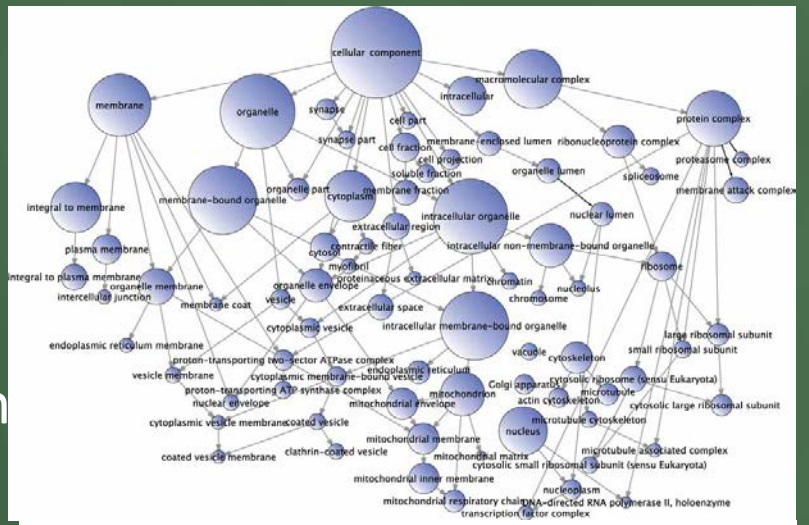
- How about text?
  - Parsegraphs

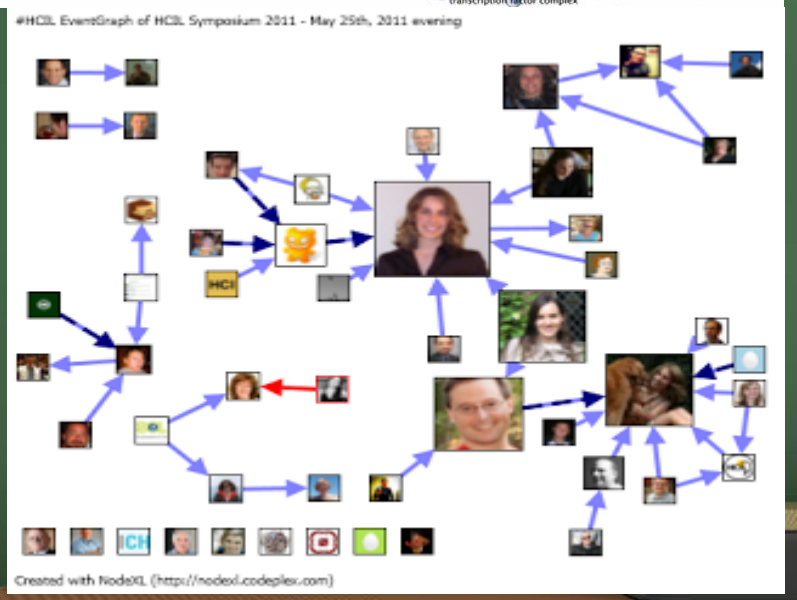


Why is a Graph a "Natural Model" for these problems?

- Ontologies are graphs (with rules)
- Social Interactions are graphs

- Therefore regardless of whether data are "born" as graphs, they can be abstracted as graphs
- **This makes graphs a uniquely positioned data model for heterogeneous information integration**
  - **However, these graphs may have different semantics that need to be accounted for**





A Real-World "Business" Problem

- Who should be on my board of advisors?
  - I already have A, B, C and D and need two more people who should
    - Have name recognition in their fields,
      - which should be "around" Computer Science
    - Should have reasonably high visibility
    - Should be known to at least two of my current members
    - Should get along with C
    - Have a lot of "business connections"
    - Be independently wealthy, and if possible, an entrepreneur
    - Not be involved in any recent negative press

Query their citation networks to compute h-index variants

Query DBPedia graph for subjects and areas related to CS, authors from DBLP to get pub venues and authors

Web pages (journals, conf., research labs …) to find "important positions"

Co-presence graphs

Ask C

Linked-In Connections

Linked-In Profile, Services sold by banks

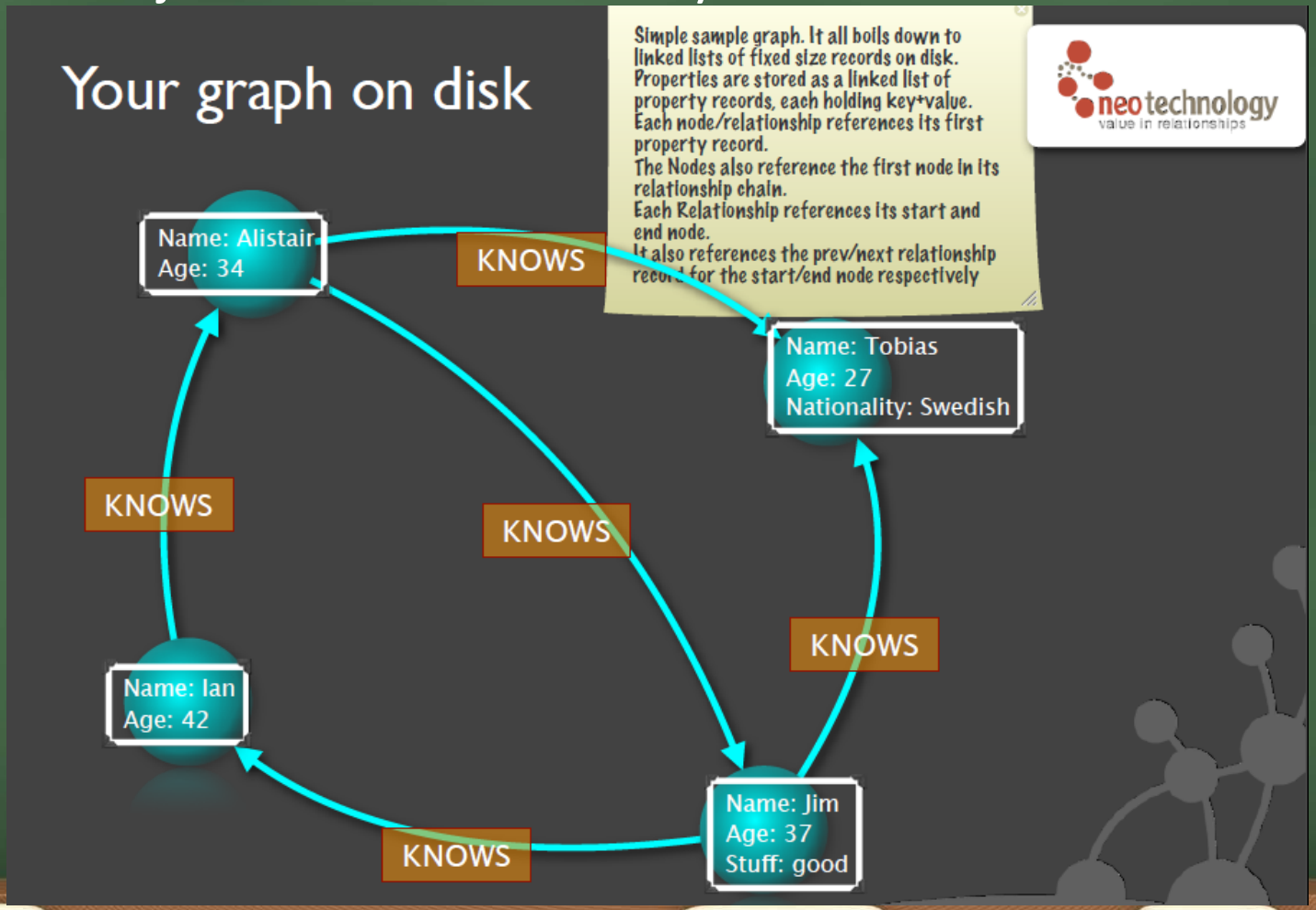
Text Analysis

Graph Functionality needed

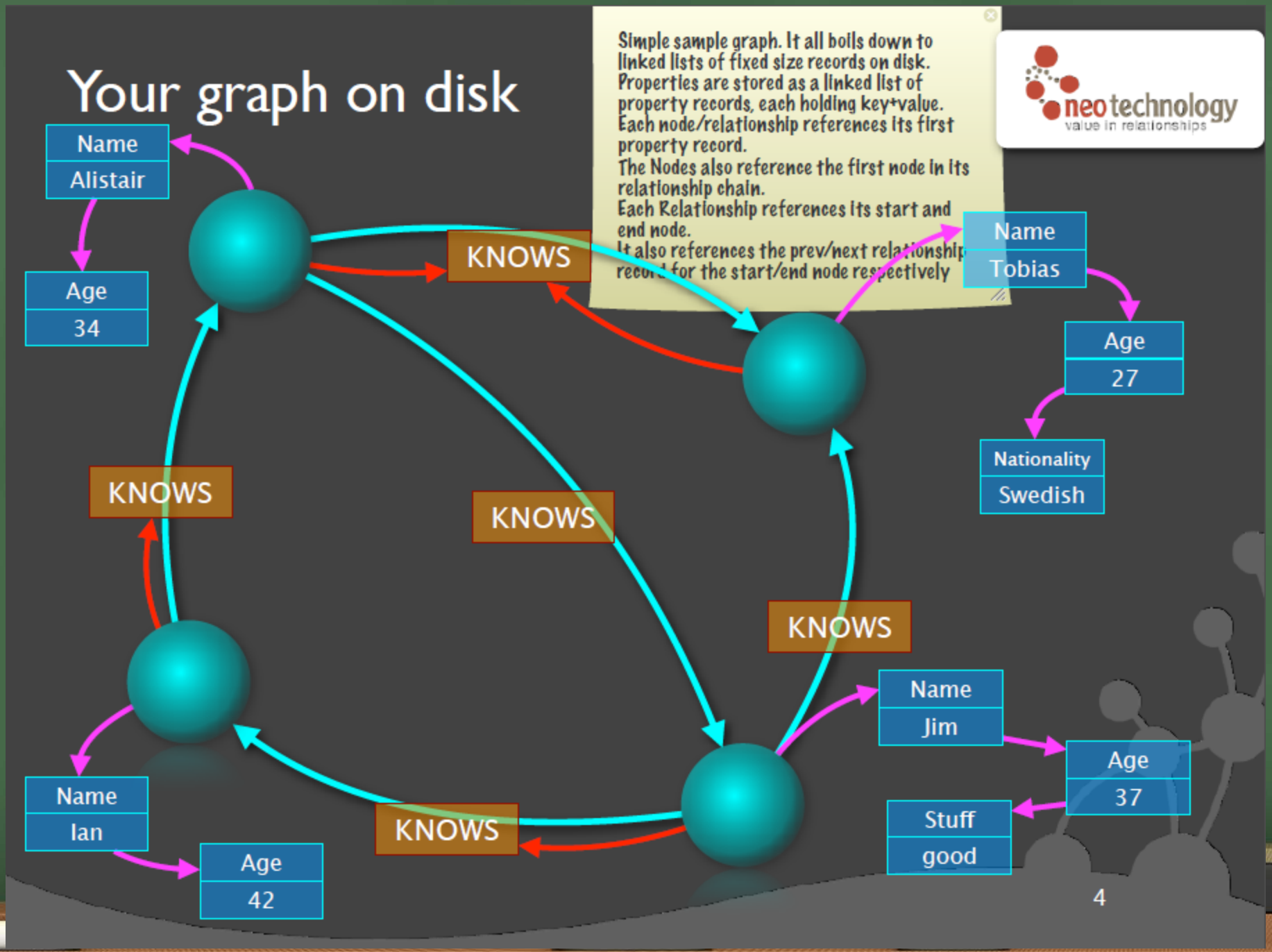
- Storage, of course
- Computation with graphs
  - For example, finding centrality measures
- Retrieval
  - Conditional traversals, query pattern matching
- Manipulations
  - Intersections, joins
- Mining
  - Finding k most frequently referred entities over a set of entity-mapped graphs in a given context
    - Finding frequent structural patterns
- Ranking over paths
  - Is this connection (i.e., path) between two members more important than that?

Representation and Storage

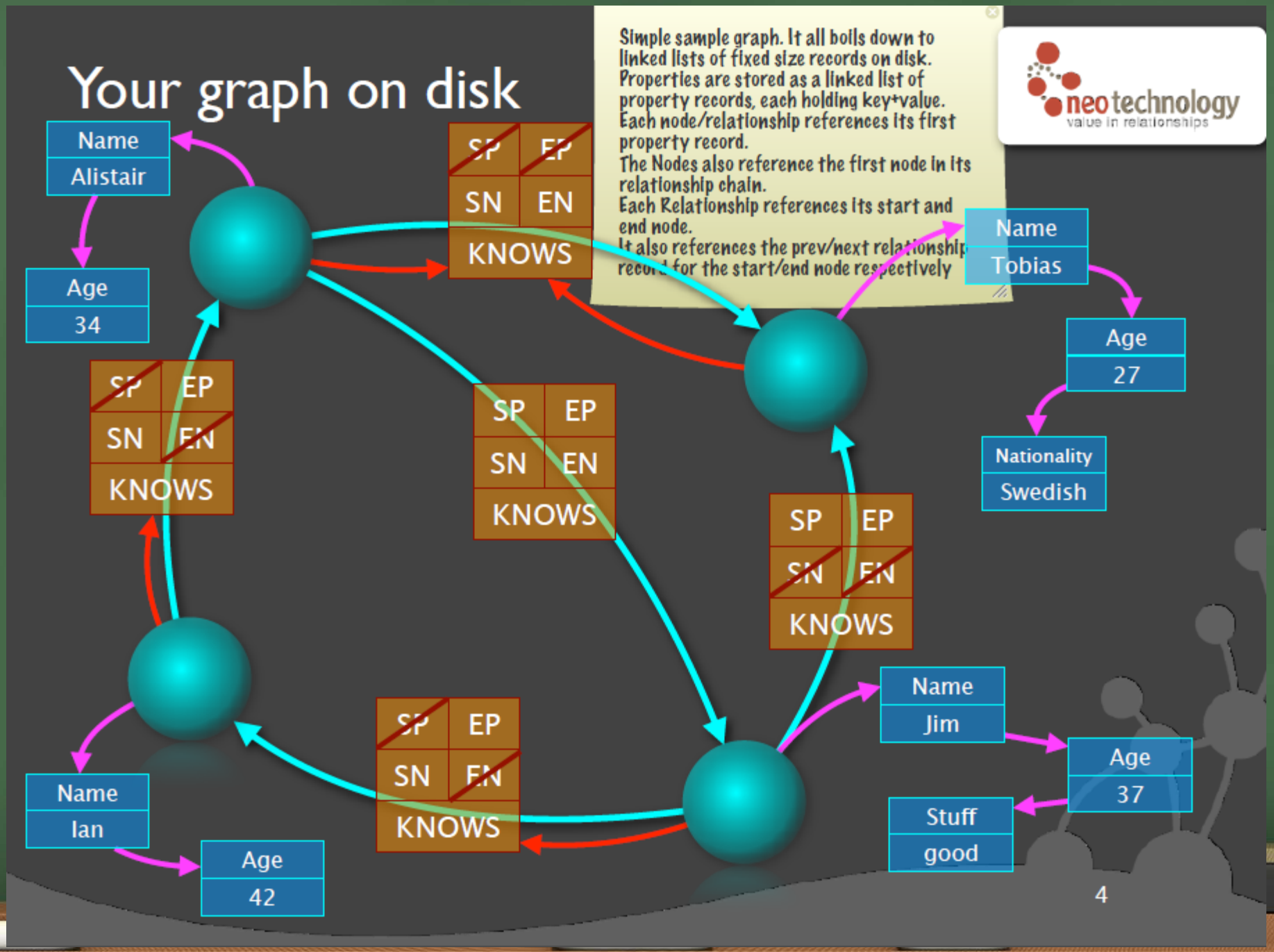
- Somewhat dependent on the intended functionality
- Neo4j – a traversal-centric system



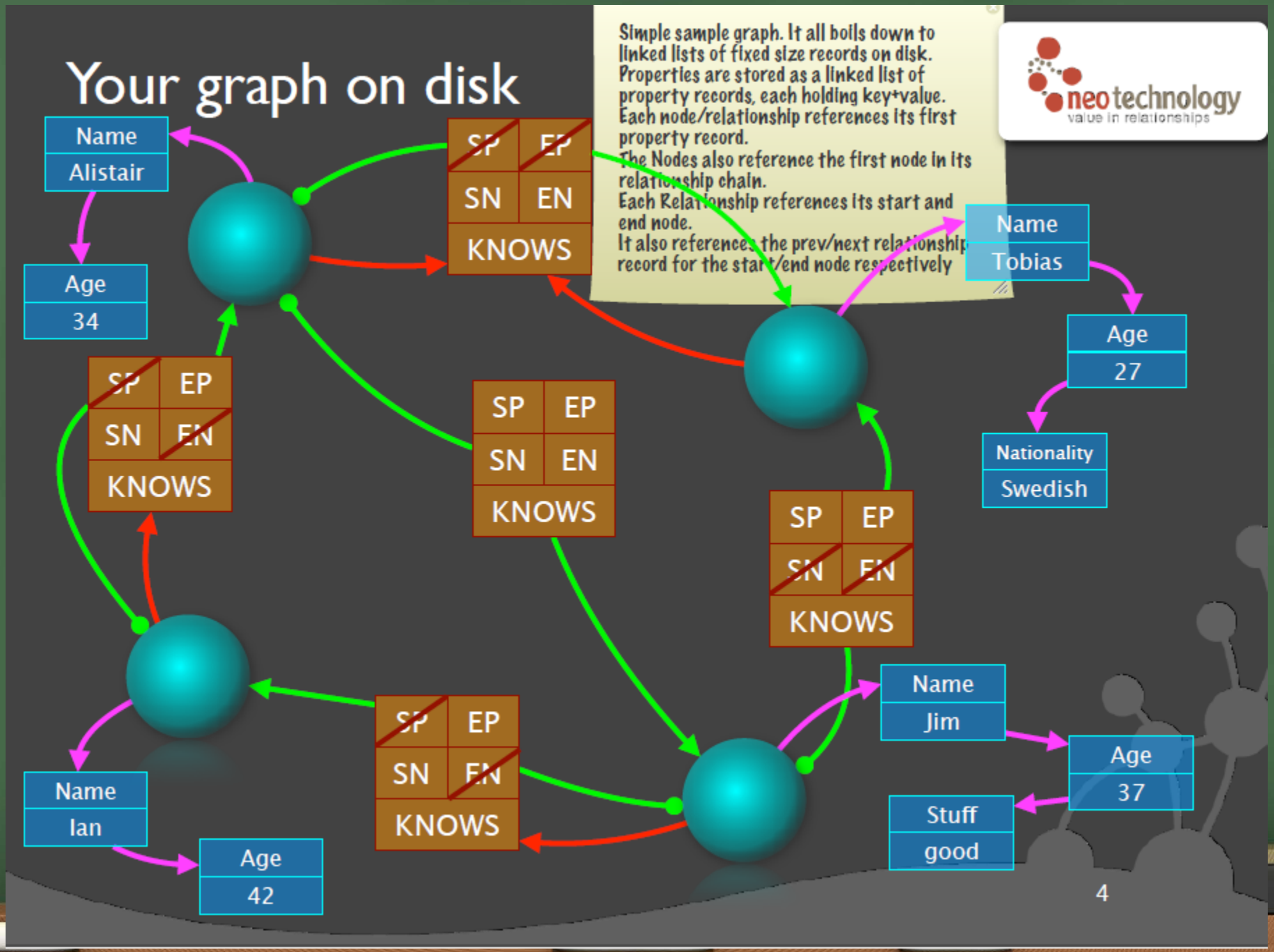
Your graph on disk



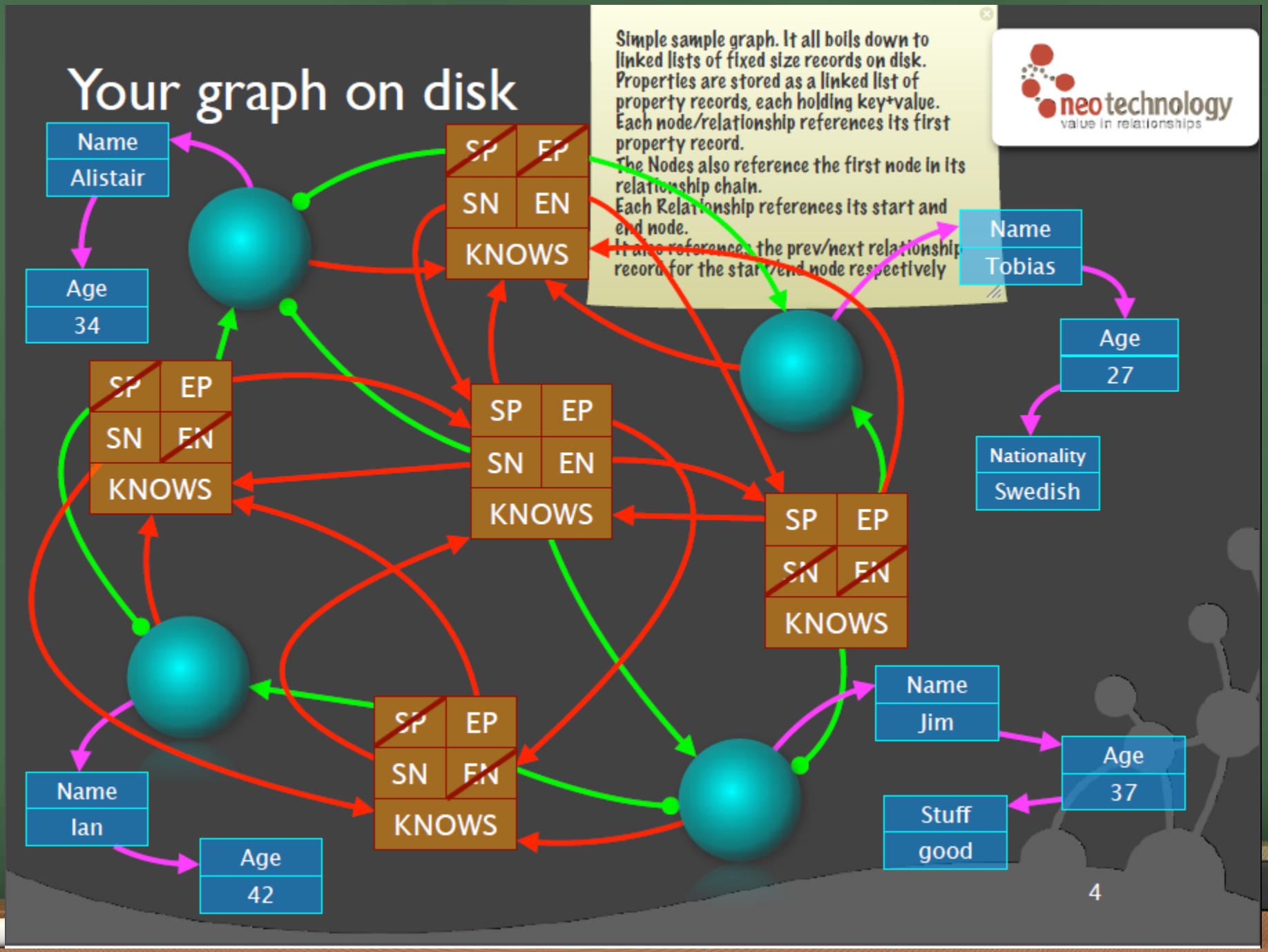
Your graph on disk



Your graph on disk

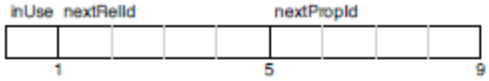


Your graph on disk

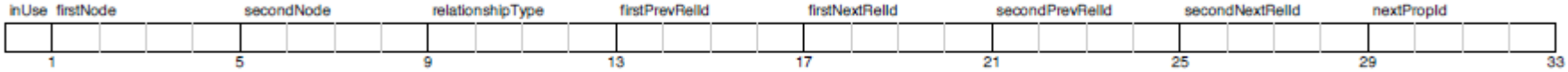


Neo4j Storage Record Layout

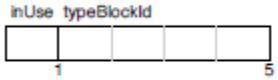
Node (9 bytes)



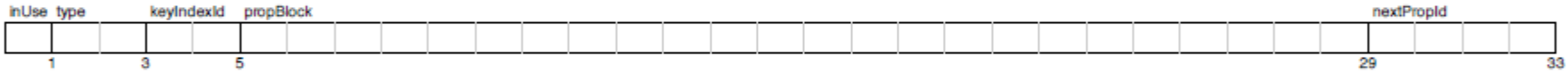
Relationship (33 bytes)



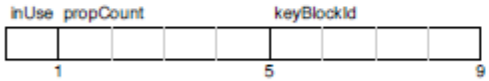
Relationship Type (5 bytes)



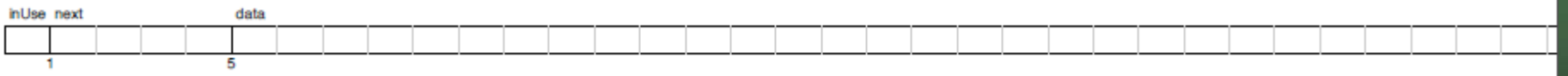
Property (33 bytes)



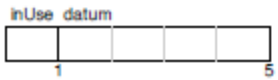
Property Index (9 bytes)



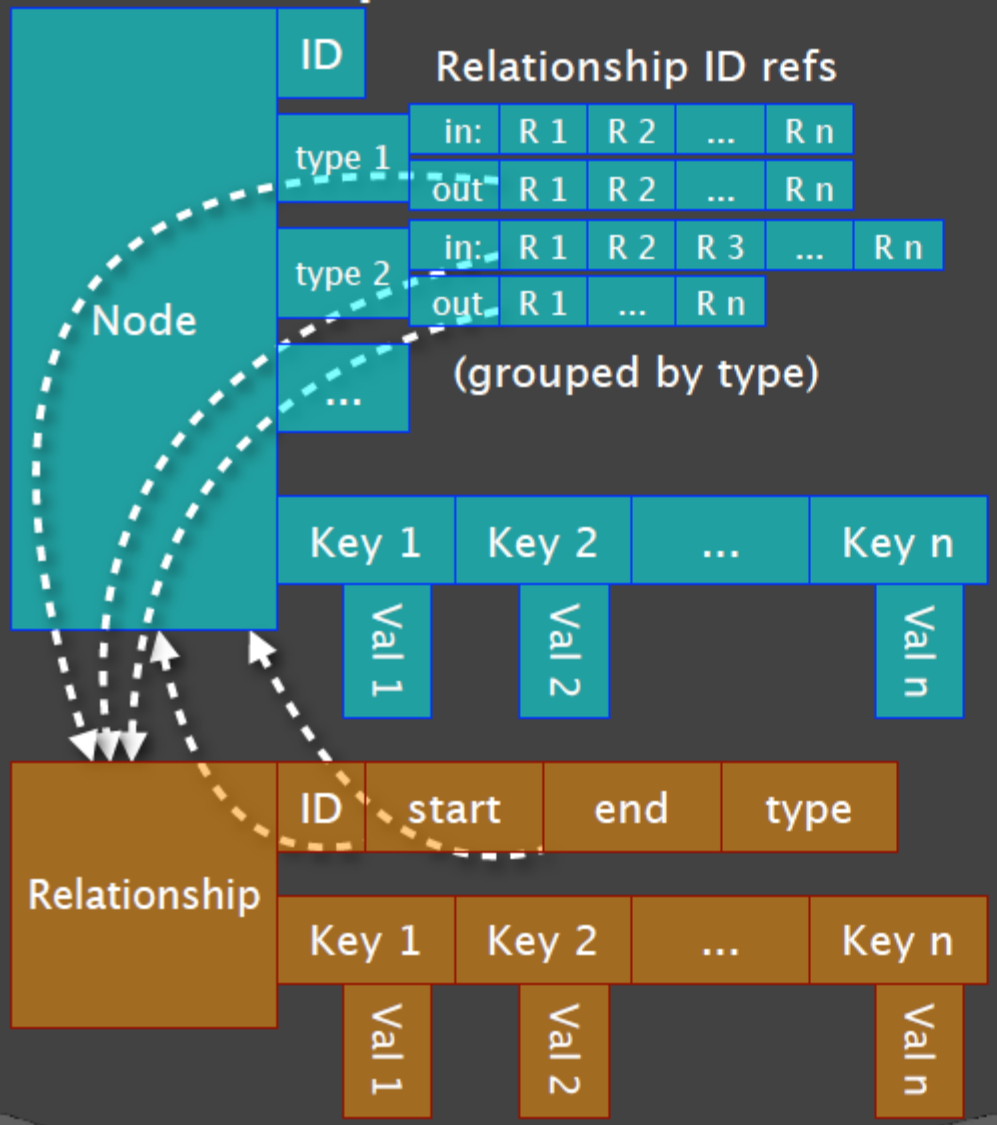
Dynamic Store (125 bytes)



NeoStore (5 bytes)



What we put in cache



The structure of the elements in the high level object cache.

On disk most of the information is contained in the relationship records, with the nodes just referencing their first relationship. In the cache this is turned around: the nodes hold references to all its relationships. The relationships are simple, only holding its properties.

The relationships for each node is grouped by RelationshipType to allow fast traversal of a specific type.

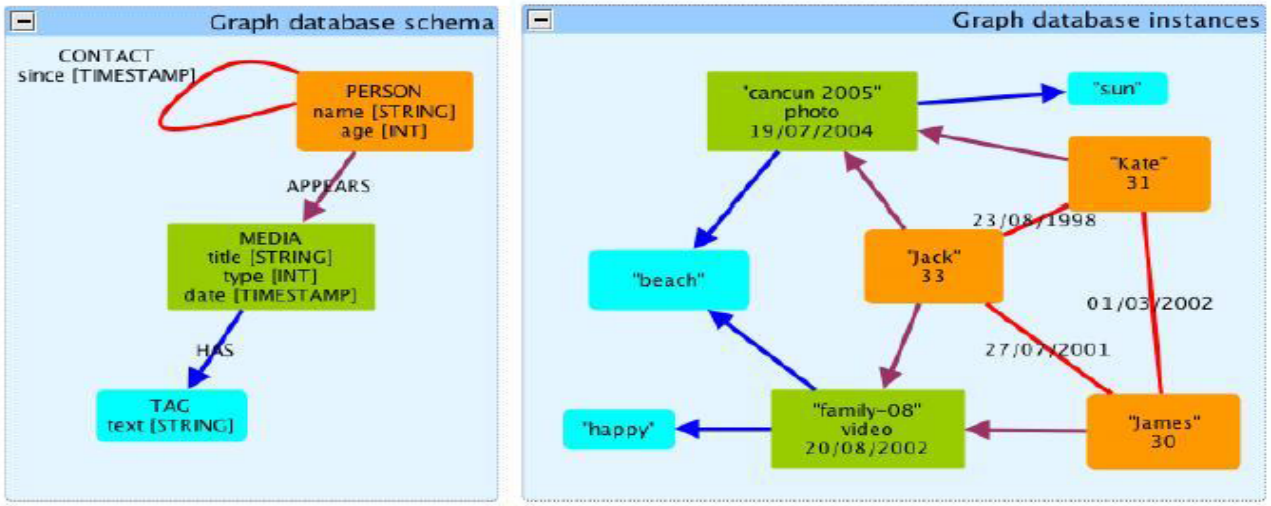
All references (dotted arrows) are by ID, and traversals do indirect lookup through the cache.

Representation and Storage

Dex: A Retrieval-Centric Storage Model

Logical graph model

- **Labeled**: nodes and edges are "typed"
- **Directed**: edges can have a fixed direction
- **Attributed**: nodes and edges can have multiple single-valued attributes
- **Multigraph**: two nodes can be connected by multiple edges

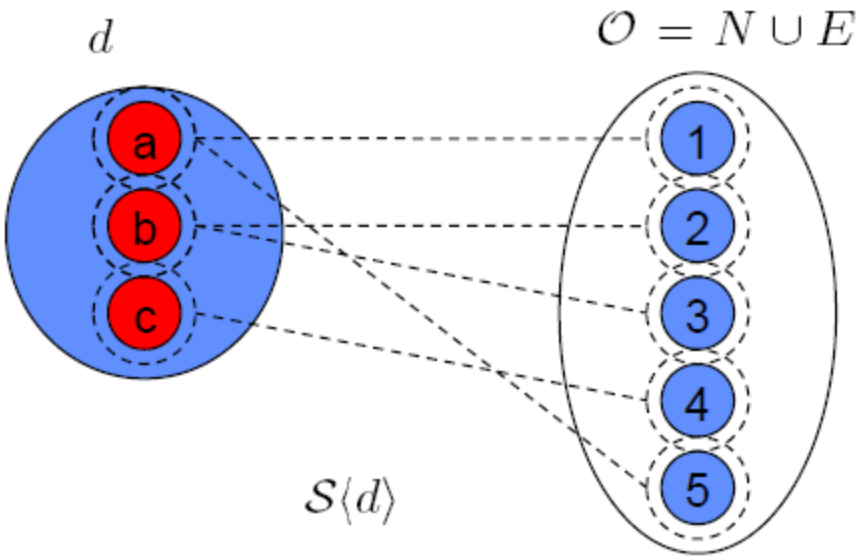


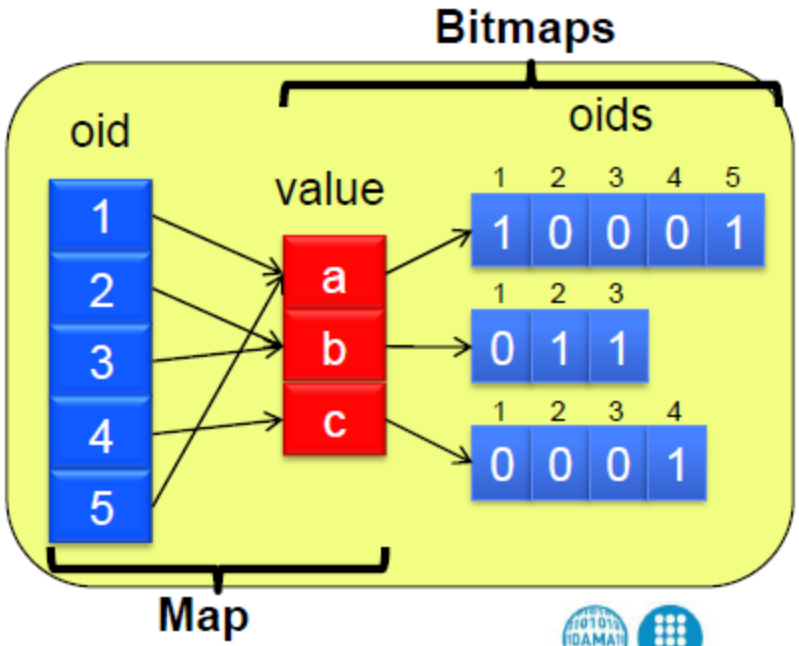
Internal representation

- Requirements
  - Split the graph into smaller structures
    - Favour the caching
    - Move to main memory just significant parts
  - OIDs instead of objects
    - Reduce memory requirements
  - Specific structures to improve traversals
    - Index edges of a node
  - Attributes fully indexed
    - Improve queries based on value filters

Internal representation

- Our approach:
  - Map + Bitmaps → Link
- Link: bidirectional association between values and OIDs
  - Two functionalities:
    - Given a value → a set of OIDs (a bitmap)
    - Given an OID → the value





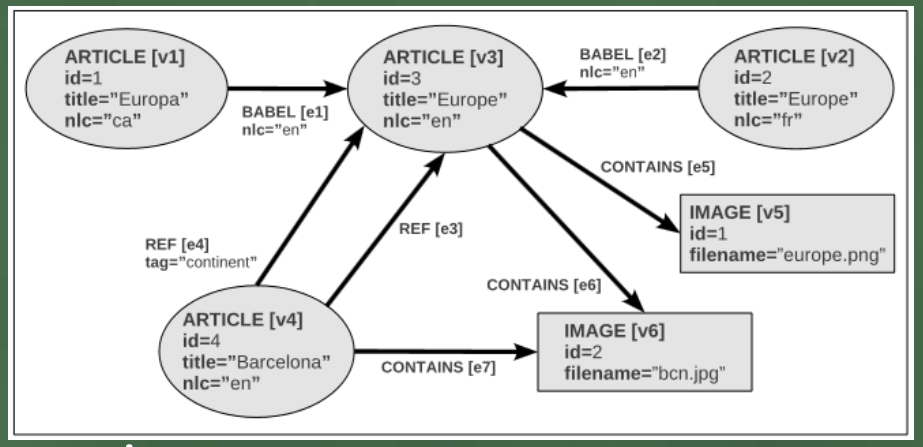
Link

sparsity technologies
performance in action

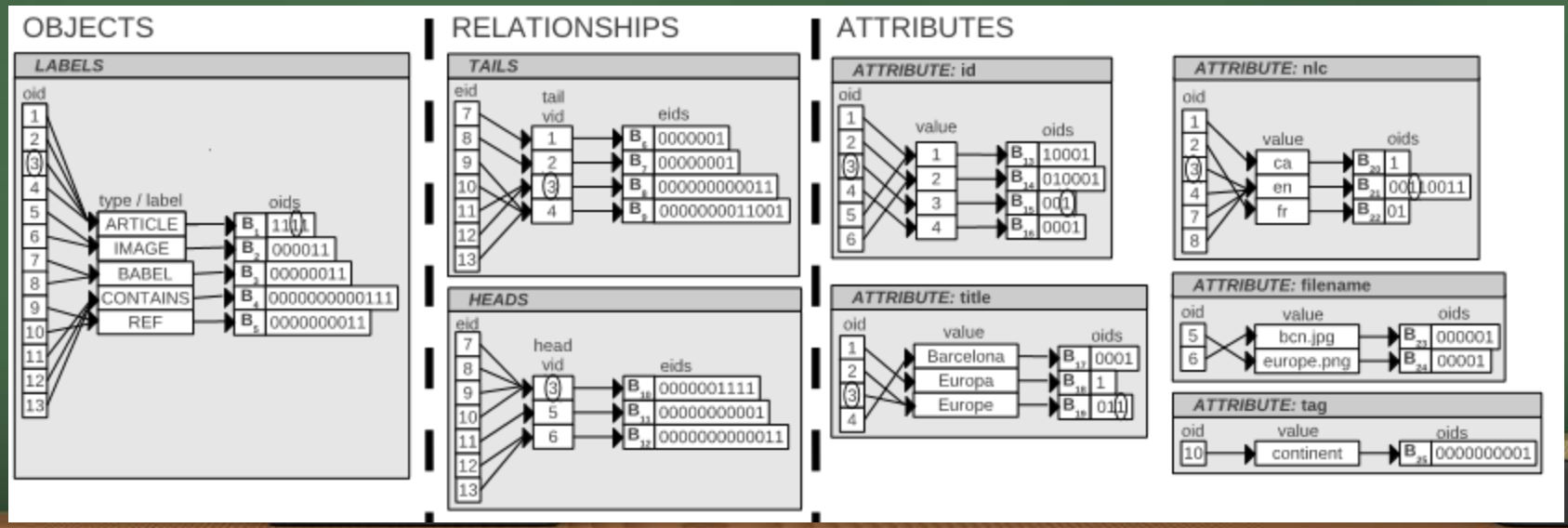
DAMA-UPC. DATA MANAGEMENT
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Graphs and Bitmaps

- A Graph



- A bitmap-based representation



Software architecture

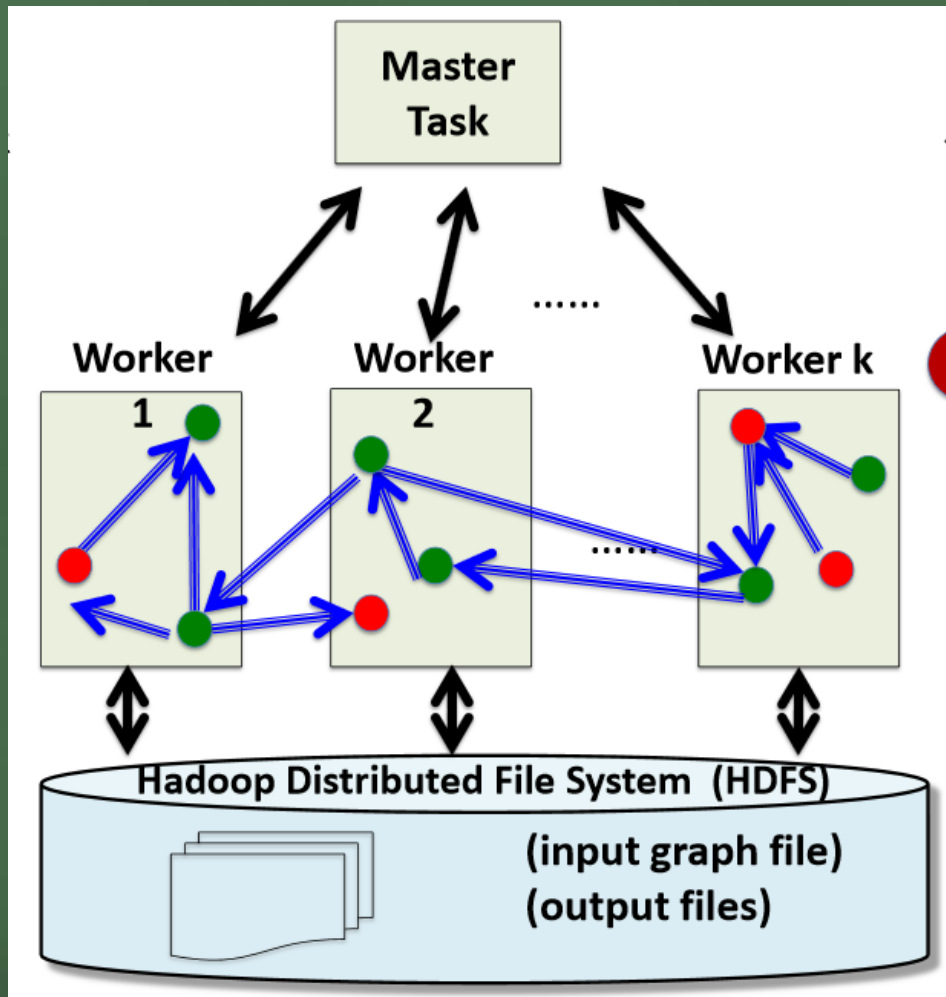
Implementation details:

- 37-bit unsigned integer OIDs
  - + 137 billion objects per graph
- Bitmaps are compressed
  - Clusters of 32 consecutive bits
  - Just existing clusters are stored
- Groups of OIDs for each type
  - Higher density of consecutive bits into bitmaps
- Maps are B+ trees
  - A compressed UTF-8 storage for UNICODE strings

Graph Computation

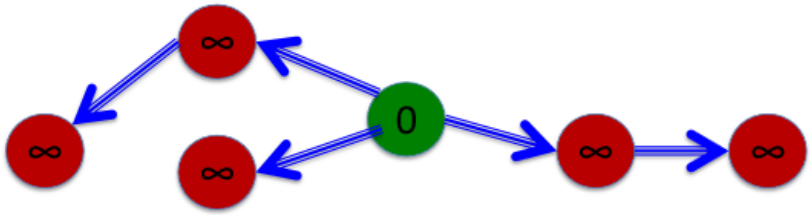
- Computing properties of nodes that are based on
  - the structure/content of the graph
  - evolving structure/content of the graph
  - Often uses adjacency matrices
- Many of these computations are iterative which eventually converge
  - Classical MapReduce-based computations are not iterative
  - Systems like Mesos and Spark are trying to modify these computations to allow iterative algorithms that pass data from iteration to iteration
    - Harder for large graphs if they don't remain in memory
- This led to the development of Bulk Synchronous Graph Processing algorithms
  - Google's Pregel

BSP Example



Pregel/GPS Overview

- Each vertex has a ***value***, and is either ***active*** or ***inactive***



- Each iteration (*superstep*): each vertex receives messages, calls a UDF *Vertex.compute()*, sends messages.
- Master calls *Master.compute()* once
- Vertex.compute()=>parallel computations
  Master.comptute()=>serial computations
- Vertices synchronize at the end of each superstep => Bulk-Synchronous Parallelism

GPS – Stanford's Graph Computation System

- Some interesting decisions
  - GPS includes an optimization called LALP (large adjacency list partitioning) where adjacency lists of high-degree vertices are partitioned across workers
    - This optimization can improve performance, but only for algorithms with two properties:
      - Vertices use their adjacency lists (outgoing neighbors) only to send messages and not for computation
      - If a vertex sends a message, it sends the same message to all of its outgoing neighbors
  - Dynamic Repartitioning
    - Reassign certain vertices to other workers dynamically during algorithm computation

Query Operations over Graphs

Neo4j

Traversals - how do they work?

neo technology
value in relationships

The surface layer, the you interact with.

- ***RelationshipExpander****s*: given (a path to) a node, returns Relationships to continue traversing from that node
- ***Evaluator****s*: given (a path to) a node, returns whether to:
  - Continue traversing on that branch (i.e. *expand*) or not
  - Include (the path to) the node in the result set or not
- Then a projection to **`Path`**, **`Node`**, or **`Relationship`** applied to each Path in the result set.

... but also:

- ***Uniqueness*** *level*: policy for when it is ok to revisit a node that has already been visited
- Implemented on top of the Core API

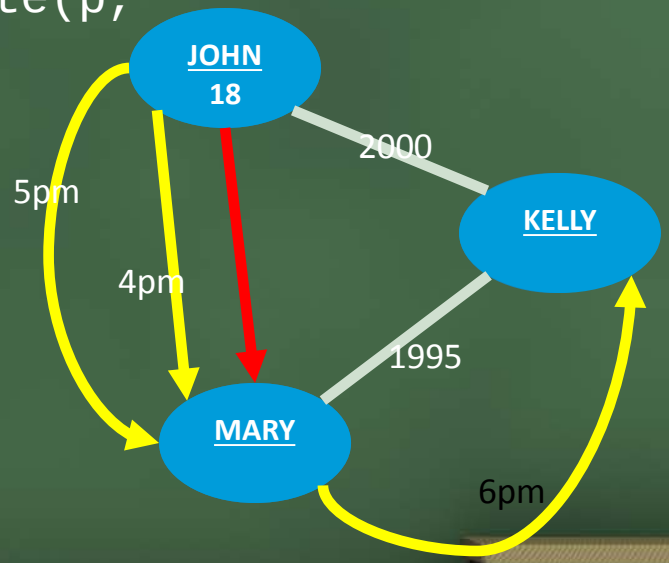
- Fetch node data from cache - non-blocking access
  - **`If`** not in cache, retrieve from storage, into cache
    - **`If`** region is in FS cache: ***blocking*** but short duration access
    - **`If`** region is outside FS cache: ***blocking*** slower access
- Get relationships from cached node
  - **`If`** not fetched, retrieve from storage, by following chains
- **`Expand`** relationship(s) to end up on next node(s)
  - The relationship knows the node, no need to fetch it yet
- **`Evaluate`**
  - possibly emitting a **`Path`** into the result set
- Repeat

This is what happens under the hood.

Query Database in Dex

Retrieve data example

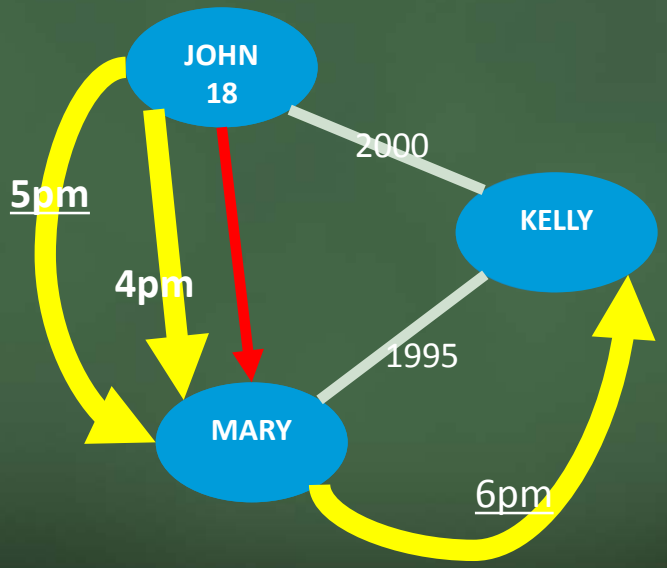
```
…
DbGraph dbg = s.getDbGraph();
Objects persons = dbg.select(person);
Objects.Iterator it = persons.iterator();
while (it.hasNext()) {
        long p = it.next();
        String name = dbg.getAttribute(p,
name).toString();
}
it.close();
persons.close();
…
```



Query Database in Dex

Navigation & Objects operations example

```
…
Objects objs1 = dbg.select(when, >=, 5pm);
// objs1 = { e5, e6 }
Objects objs2 = dbg.explode(p1, phones, OUT);
// objs2 = { e4, e5 }
Objects objs = objs1.intersection(objs2);
// objs = { e5, e6 } ∩ { e4, e5 } = { e5 }
…
objs.close();
objs1.close();
objs2.close();
…
```



OUR RESEARCH – OPERATING ON ONTOLOGY GRAPHS

The Shredded Ontology - 1

- The reified triples

```
<owl:Class rdf:ID="Wine">
 <rdfs:subClassOf rdf:resource="&food;PotableLiquid" />
 <rdfs:subClassOf>
    <owl:Restriction>
          <owl:onProperty rdf:resource="#hasMaker" />
          <owl:someValuesFrom rdf:resource="#Winery" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

$\text{Wine} \sqsubseteq \text{food:PortableLiquid}$
$\text{Wine} \sqsubseteq \exists\ \text{hasMaker.Winery}$

```
<Wine>   rdf:type          owl:Class
<Wine>   rdfs:subClassOf    food:PotableLiquid
<Wine>   <hasMaker>         <Winery>
<Wine>   rdfs:subClassOf    <something [exists] hasMaker Winery>
<something [exists]hasMaker Winery> rdf:type          owl:Restriction
<something [exists]hasMaker Winery> owl:onProperty    <hasMaker>
<something [exists]hasMaker Winery> owl:someValuesFrom <Winery>
```

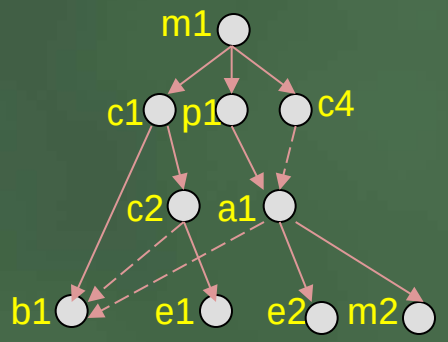
Locally inferred triples

Syntax-based triples

The Shredded Ontology – 2

DAGs for
- transitive relationships
  - part-of, continuous-with, member-of
- sub-property relationships
  - volumetric-part-of $<_{sp}$ proper-part-of $<_{sp}$ part-of

- The DAG Indexes
  - One per transitive relationship, one per subproperty tree
  - Modified SSPI
    - Index the embedded tree and non-tree portions separately
      - Embedded tree has a Dewey index
      - Non-tree edges is maintained in a "minimal" skeleton structure to connect them to nearest ancestors
  - Statistics is kept at nodes to perform limited-depth queries
  - Complex, multi-reachability patterns still a problem
    - It is wiser to treat "hub nodes" specially



nid	preds
b1	{c2, a1}
a1	{c4}
e2	{a1}
m2	{a1}

The Shredded Ontology – 3

- Bitmap Indices
  - Derived from RDF triples

PSIndex	Property	Subject	Objects (bitmap)
POIndex	Property	Object	Subjects (bitmap)
SOIndex	Subject	Object	Properties(bitmap)

SSJIndex	Property	Property	Subjects (bitmap)	(S,P1,O1),(S,P2,O2)
SOJIndex	Property	Property	Subjects (bitmap)	(X,P1,O),(S,P2,X)
OOJIndex	Property	Property	Objects (bitmap)	(S1,P1,O),(S2,P1,O)

- Using the Bitmap Indices
  - Select genes that have no associated proteins
    - POIndex(type,gene) && ! SSJoin(type,expressesProtein)

The Shredded Ontology – 4

- The keyword index
  - Simple inverted index of all string-valued literals
  - Support partial string matches and regular expressions on strings
  - Distinguish between class nodes, edge labels and instance nodes

Keyword Queries using the Ontology

- Classify keywords
  - Find LCA concepts of a conjunctive query
  - Find if specific distinguished classes appear in queries
    - "Alzheimer's" subclass-of Disease
- Apply Class-specific expansion rules
  - For items classified as anatomical object get part-of descendants, not including the cell module
  - For items classified as cell, get subclasses, by executing property chains if needed
    - property chain is a new edge-label, defined using a positive, non-recursive first order rule
- Find data in sources using expanded query
- Ontological relationships
  - Find up to k-distance paths amongst pairs of *hot keywords* in conjunctive queries
  - Find data source elements that are mapped to these relationships
- Rank Results??

Querying the Ontology

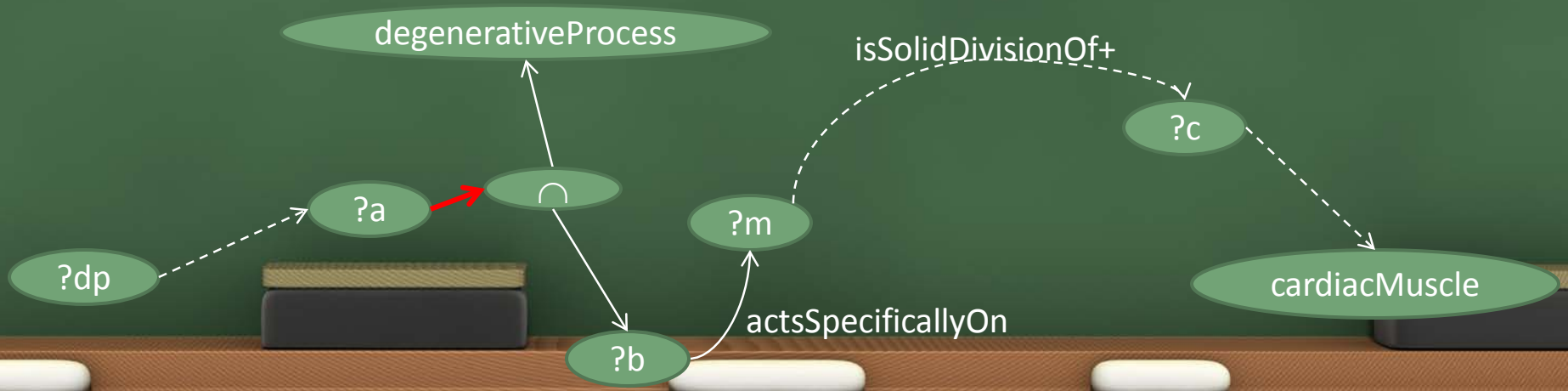
- Extending the TERP query language (Sirin et al)
  - TERP is a syntactic enhancement of the SPARQL
    - Our extensions allow
      - transitive operations and path expressions on edges
      - graph output

```
SELECT ?diseaseProcess
WHERE {
        ?diseaseProcess rdfs:subClassOf+ (:degenerativeProcess and
                                          :actsSpecificallyOn some ?muscle) .
                                          (?muscle :isSolidDivisionOf+
                                                   ( :subClassOf+ cardiacMuscle) )
}
```

Query Planning within OntoQuest

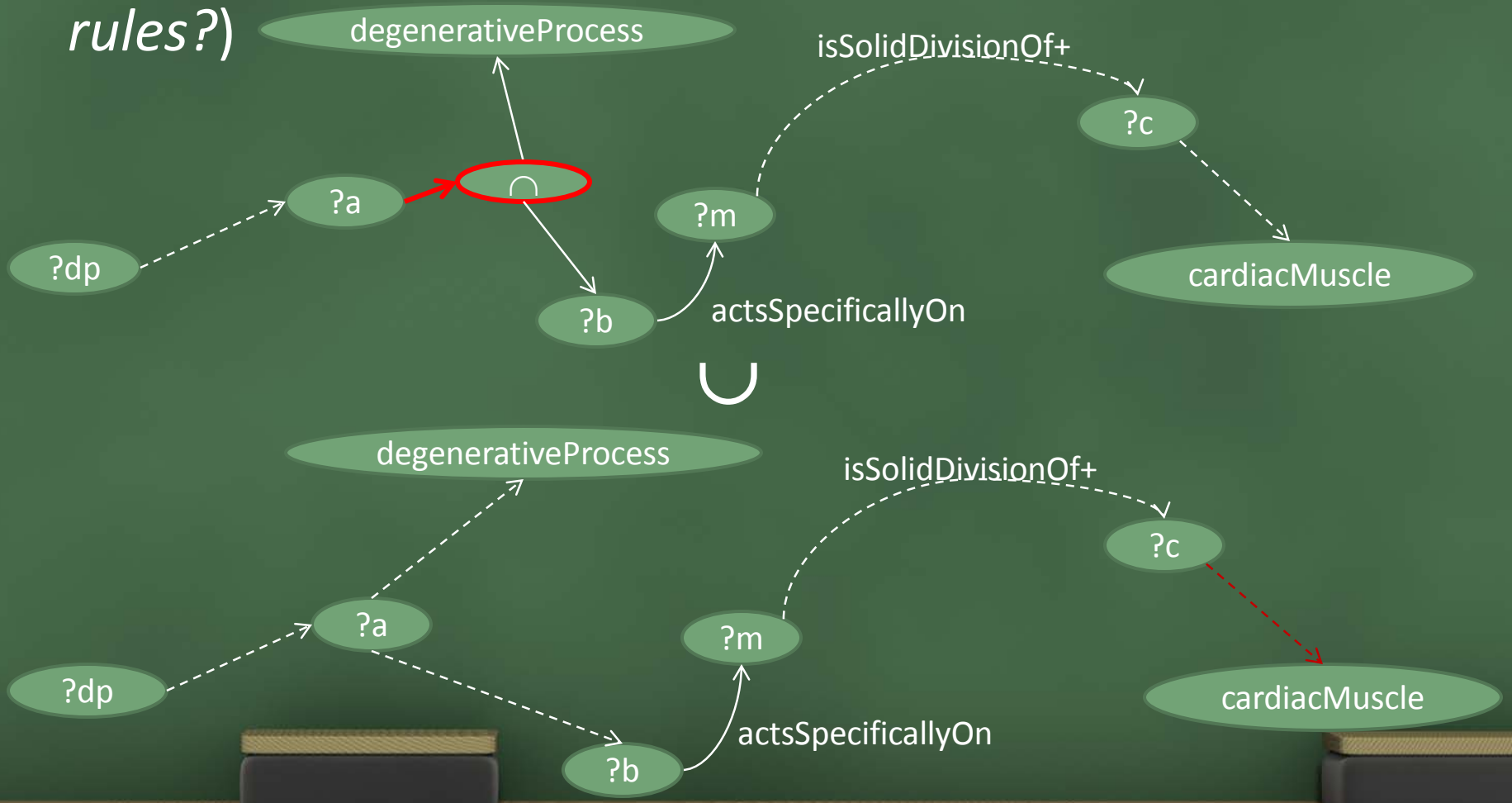
- Rewrite the **where** clause using intermediate variables

```
WHERE {
        ?diseaseProcess rdfs:subClassOf+ ?a .
        ?a intersectionOf (:degenerativeProcess, ?b) .
        ?b onProperty :actsSpecificallyOn .
        ?b owl:someValuesFrom ?muscle .
        ?muscle :isSolidDivisionOf+ ?c .
        ?c rdfs:subClassOf+ :cardiacMuscle
}
```



Query Planning within OntoQuest

- Some semantic rewrites (*what are all the rewrite rules?*)



Query Planning and Optimization

- Some standard rewrites
  - Map GRAPH IRI GroupGraphPattern to Graph(IRI, GroupGraphPattern)
  - Map all graph patterns contained in a group to produce a list, SP, of algebra expressions
    - For example
      - for i := 0 ; i < length(SP); i++
      - If SP[i] is an OPTIONAL,
      - If SP[i] is of the form OPTIONAL(Filter(F, A))
      - G := LeftJoin(G, A, F)
      - else
      - G := LeftJoin(G , A, true)
      - Otherwise for expression SP[i], G := Join(G, SP[i])

Selectivity

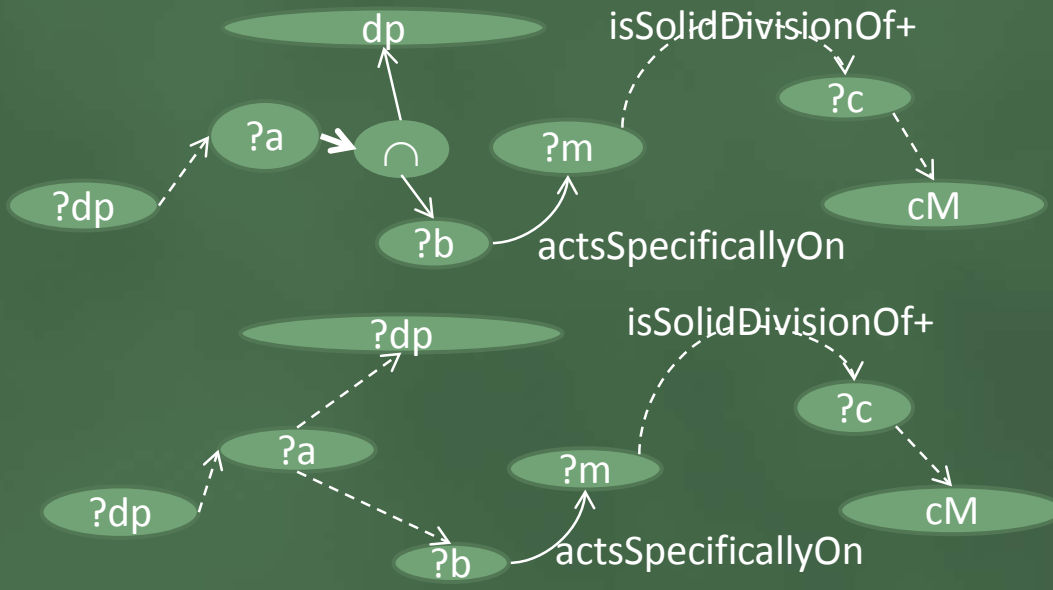
- Basic Graph Patterns are planned by selectivity estimates

#	Bound Variables	Operation Type	Selectivity
1	prop1, prop2, obj	SSJoin	High
2	prop1, prop2, obj	SOJoin	Very high
3	prop1, sub, prop2	OOJoin	High
4	prop1, prop2	SOJoin, OOJoin or SSJoin	Medium
5	Sub, obj	SOSelect	High
6	Sub, prop	PSSelect	High
7	Prop, obj	PSSelect	High
8	prop	Union(PSSelect), Union(POSelect)	Very Low
9	sub	Union(PSSelect), Union(SOSelect)	Low
10	obj	Union(POSelect), Union(SOSelect)	Low
11	Prop+, obj	DAG_descendants	High
12	Prop+, sub	DAG_ancestors	Medium
13	keyword	Kw_search	High

Next round should use statistics and a cost model

Query Plan Example

- Query




```
?c = DAG_descendants('rdfs:subClassOf', kw-index(:cardiacMuscle));
?muscle = DAG_descendants(':isSolidDivisionOf', ?c);
?b = POSelect(?b kw_index(:actsSpecificallyOn) to_bitmap(?muscle));
?a1= AND (:dP, ?b);
?a2=BITMAP_AND (sojoin(sc+, sc+), to_bitmap(dag_desc(sc, :dP)))
?a = ?a1 union ?a2
?diseaseProcess = DAG_descendants('rdfs:subClassOf', ?a).
```

OntoQuest Performance – 1

- Data set 1: NIFSTD ontology
  - # of Nodes: 467,848, # of Nodes in the subClassOf DAG: 45,882, total # of descendant results:

Metric	Time (in ms)
Avg. time for getDescendants	12.21
Max. time for getDescendants	1866 (203,222 results)
Avg. time to get paths between 2 nodes	28
Avg. time to get all paths to a node through k given nodes	190
Given an unordered list of nodes, find the paths connecting all of them	210
Time for SSJoin, SOJoin, OOJoin over 66,564 p-p pairs (bitmap returned)	644, 456, 468 resp.
Time for SSJoin, SOJoin, OOJoin over 66,564 p-p pairs (Node IDs returned)	977, 593, 480 resp.

OntoQuest Performance – 2

Query	Time(ms)

SELECT ?structure
WHERE {
?structure rdfs:subClassOf ?o1 .
?o1 owl:intersectionOf (:SolidStructure ?a2 ?a3) .
?a2 owl:onProperty :isStructuralComponentOf .
?a2 owl:someValuesFrom :Leg .
?a3 owl:onProperty :isPairedOrUnpaired .
?a3 owl:someValuesFrom :atLeastPaired
}
	175

SELECT ?structure WHERE { ?structure rdfs:subClassOf+ :SolidStructure }	1450 tuples 173(only node ids) 336(all node properties)
SELECT ?structure WHERE { ?structure rdfs:subClassOf+ (:SolidStructure and :Duct). }	209 (only node ids) 211(all node properties)
What degenerative diseases affect some cardiac muscle?	206(only node ids) 207 (all node properties)

Next Steps in our Research

- Exploring an extended version of neo4j as the graph representation
- Combining with a BSP style computation engine
- Developing architecture-aware optimization rules