Querying Data on the Web

Alvaro A A Fernandes
School of Computer Science
University of Manchester

COMP62421:: 2015-2016

Part V

Massive Distribution, Massive Parallelism

Teaching Day 5 Outline

1. Introduction
2. Querying the Web of Data
   - Data for/from the Web v. Data on the Web
   - A Web of Data
   - The Wild Wild Web'
   - Data on the Web of Data
   - The Web of (Linked) Data
   - Linked-Based Query Evaluation
3. Massively-Parallel Schemes: NoSQL
   - DBMSs, Then and Now
   - 21st-Century: Big Data
   - Scalable OLTP
   - NoSQL: Why, What, When
4. Massively-Parallel Schemes: The Map-Reduce Model
   - Cluster-Based Data Management
   - The Map-Reduce Computational Model
5. Massively-Parallel Schemes: Map-Reduce Engines
   - Motivation and Functionalities
   - Programming/Execution
   - Locality, Fault Tolerance, Optimizations
6. Query Processing with Map-Reduce
   - A Worked-Out Motivating Example
   - Map-Reduce Semantics for Relational-Algebraic Operators
7. Conclusion

Topics So Far (1)

- We have revised the basic notions of
  - what a **database management system** (DBMS) is
  - the role that is played by the languages a DBMS supports
  - how DBMS-centred applications are designed
- We have adopted an approach to describing the internal architecture of DBMSs that allowed us to discuss
  - the strengths and weaknesses of classical DBMSs
  - the recent trends in the way organizations use DBMS
  - how architectures have been diversifying recently
- We have revised the various kinds of query languages by looking into
  - the (tuple) relational calculus
  - a relational algebra
  - SQL

Topics So Far (2)

- We have looked in detail into logical optimization, i.e., heuristic-based rewriting of algebraic plans.
- We have explored some of the basic strategies for designing physical operators and for combining them into query evaluation plans (QEPs).
- We have briefly discussed some of the challenges in cost-based QEP selection, in particular the hard problem of choosing a join order.
- We have found out how data partitioning strategies, allied with the algebraic properties of query languages, make it possible to automatically parallelize QEP execution.

Topics So Far (3)

- We have taken a quick, example-driven tour of XQuery, to remind ourselves of the basic characteristics of the language.
- We have introduced and briefly explored an algebraic view of XQuery, and some equivalence laws that arise, on the basis of which one can conceive of a rewriting approach to logical optimization for the language.
- We have studied storage, optimization and evaluation of XQuery as implemented in one specific XML native DBMS, viz., BaseX.

Topics So Far (4)

- We have taken a quick, example-driven tour of SPARQL, to remind ourselves of the basic characteristics of the language.
- We have introduced and briefly explored an algebraic view of SPARQL, and some equivalence laws that arise, on the basis of which one can conceive of a rewriting approach to logical optimization for the language.
- We have studied storage, optimization and evaluation of SPARQL, specially in the context of the SPARQL-through-SQL strategy.

Learning Objectives So Far

- We have aimed to revise and reinforce undergraduate-level understanding of DBMS as software systems, rather than as software development components.
- We have aimed to get under way in our postgraduate-level exploration of query languages, in preparation for further delving into query processing techniques this week.
- We have aimed to explore classical query processing, including optimization and evaluation, both centralized and parallel.
- We have aimed to acquire a deeper understanding of some of the challenges arising in querying over large XML collections using XQuery.
- We have aimed to acquire a deeper understanding of some of the challenges arising in querying over large RDF collections using SPARQL.

This Week: Topics

- We will look into querying (live, distributed) data on the web, as opposed to querying (locally-materialized) data from/for the web.
- We will also briefly look into the NoSQL approach to data management.
- Finally, we will delve a little on map-reduce engines as a platform for query processing.

This Week: Learning Objectives

- We will aim to acquire a deeper understanding of distributed querying on the web.
- We will aim to acquire a deeper understanding of distributed stores under the NoSQL paradigm.
- We will aim to gain an insight into how map-reduce engines offer an opportunity for massively-parallelization of query execution.

Acknowledgements (1)

The material presented mixes original material by the author as well as material adapted from

- [Monash, 2007a, Monash, 2007b]
- [Stonebraker and Cattell, 2011]
- [Cattell, 2010]
- [DeCandia et al., 2007]
- [Leskovec et al., 2015]

The author gratefully acknowledges the work of the authors cited while assuming complete responsibility any for mistake introduced in the adaptation of the material.

Acknowledgements (2)

***

Portions of the material presented are modifications based on work created and shared by Google and used according to terms described in the Creative Commons 2.5 Attribution License:

http://creativecommons.org/licenses/by/2.5/

The original content was authored by Aaron Kimball, Sierra Michels-Slettvet, Christophe Bisciglia, Hannah Tang, Albert Wong, Alex Moshchuk et al. and used to be accessible but the link has now been discontinued by Google:

http://code.google.com/edu/content/parallel.html

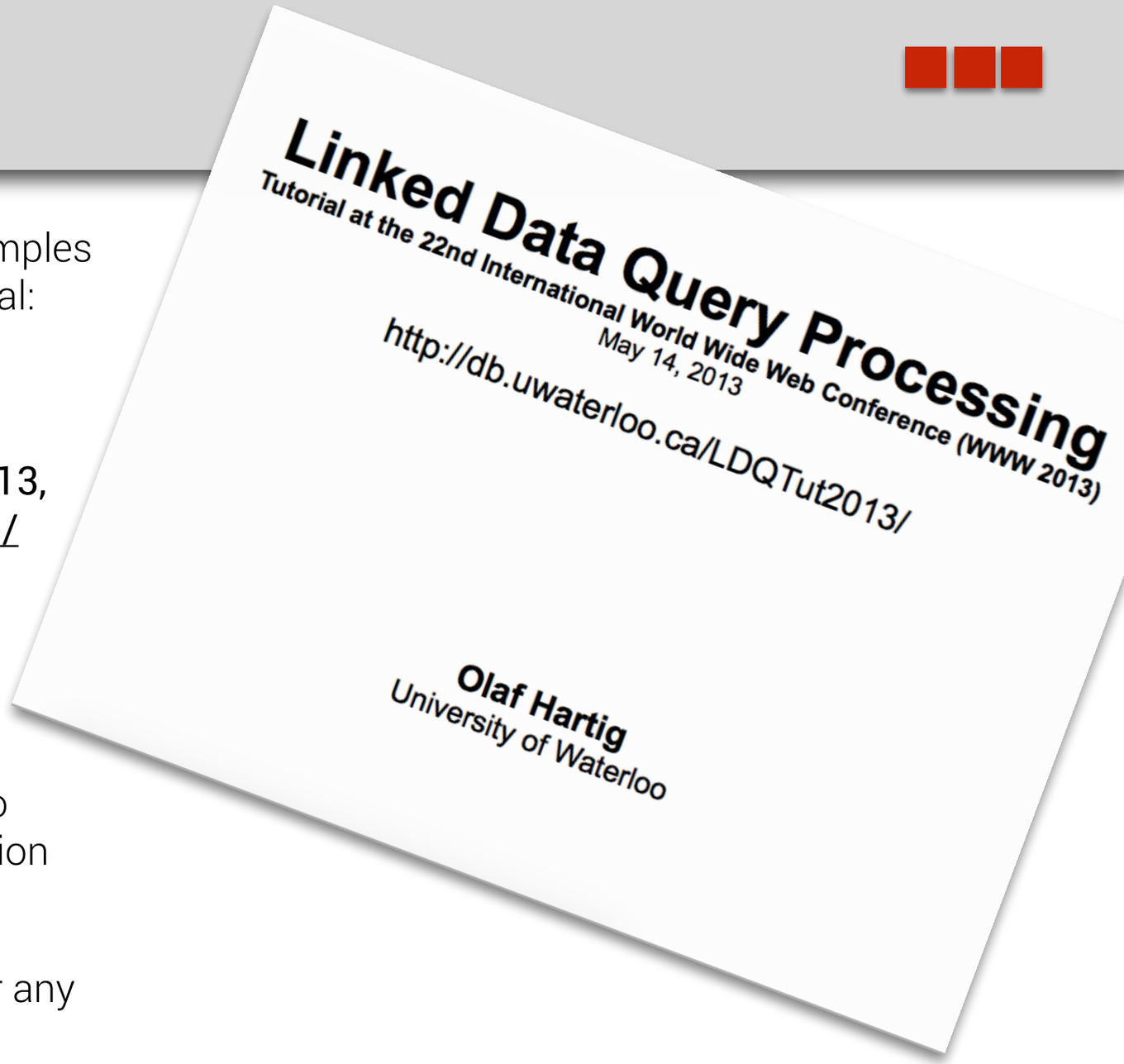
The current author is grateful to Google and the authors of the original content for sharing their work. The current author assumes responsibility any for mistake introduced in the modifications made.

***

Lecture 21

Querying the Web of Data

Acknowledgement



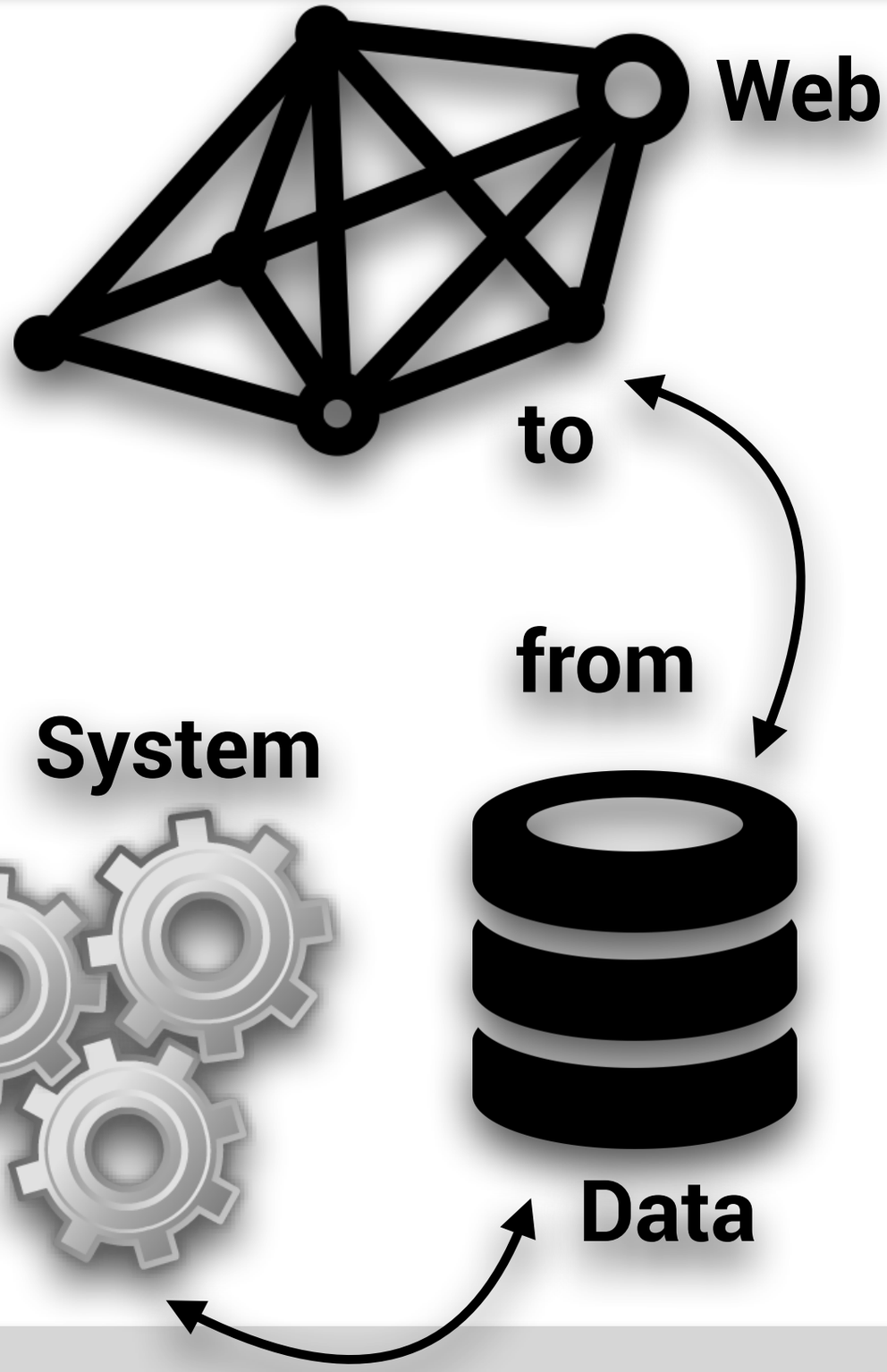
- Some of these slides mostly contain text and examples that are most often taken *verbatim* from the tutorial:
  - Hartig, Olaf (2013). Linked Data Query Processing. Tutorial at the 22th International World Wide Web Conference (WWW), May 2013, Rio de Janeiro, Brazil. http://db.uwaterloo.ca/LDQTut2013/
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect recent updates to SPARQL or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above author is acknowledged.
- The author of these slides is very grateful to the author of the work above.

MANCHESTER 1824

The University of Manchester

Data for/from the Web v. Data on the Web

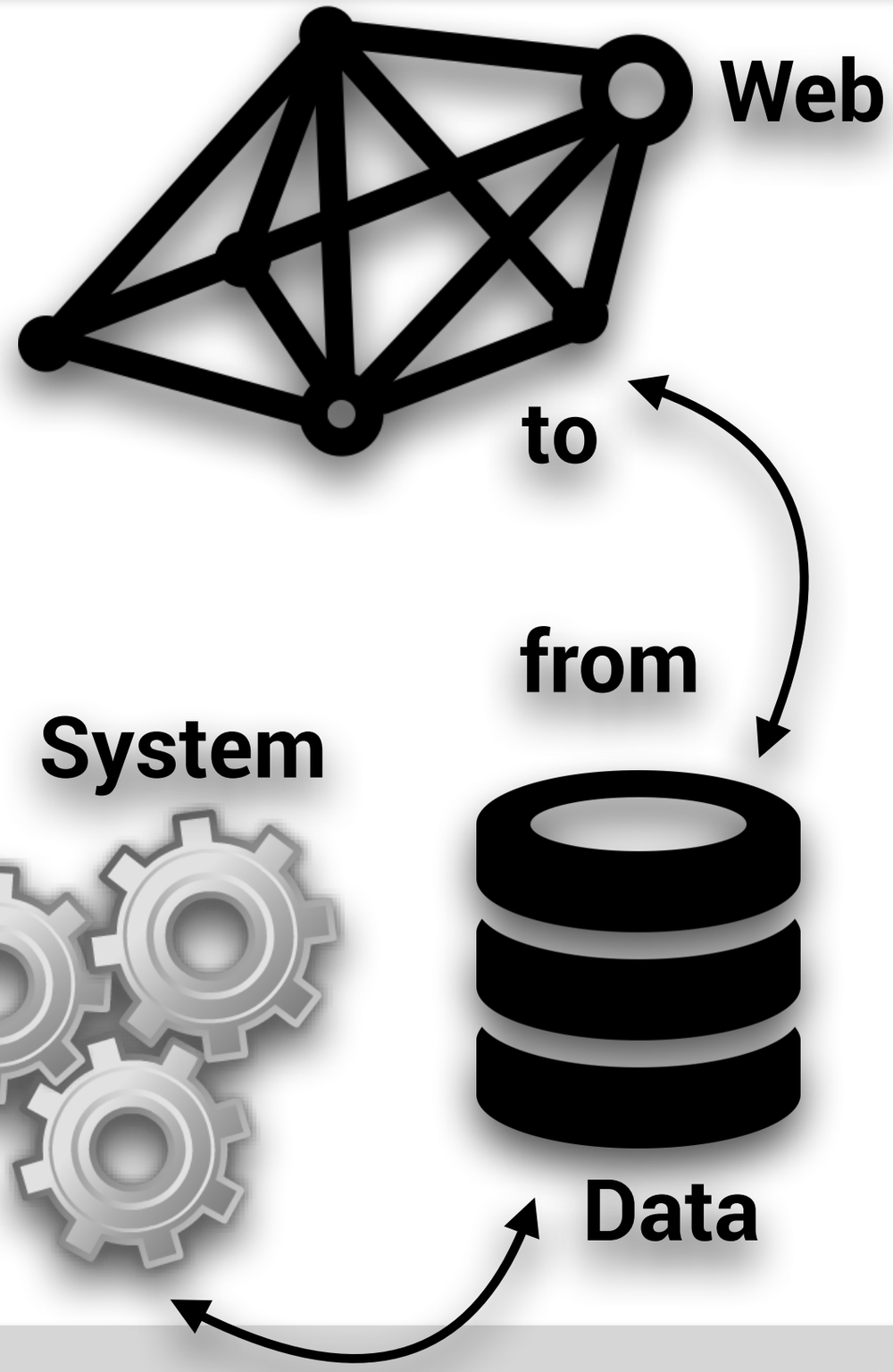
Data and the Web



- So far, we have explored not so much data **on** the Web as data **from** (or **for**) the Web.

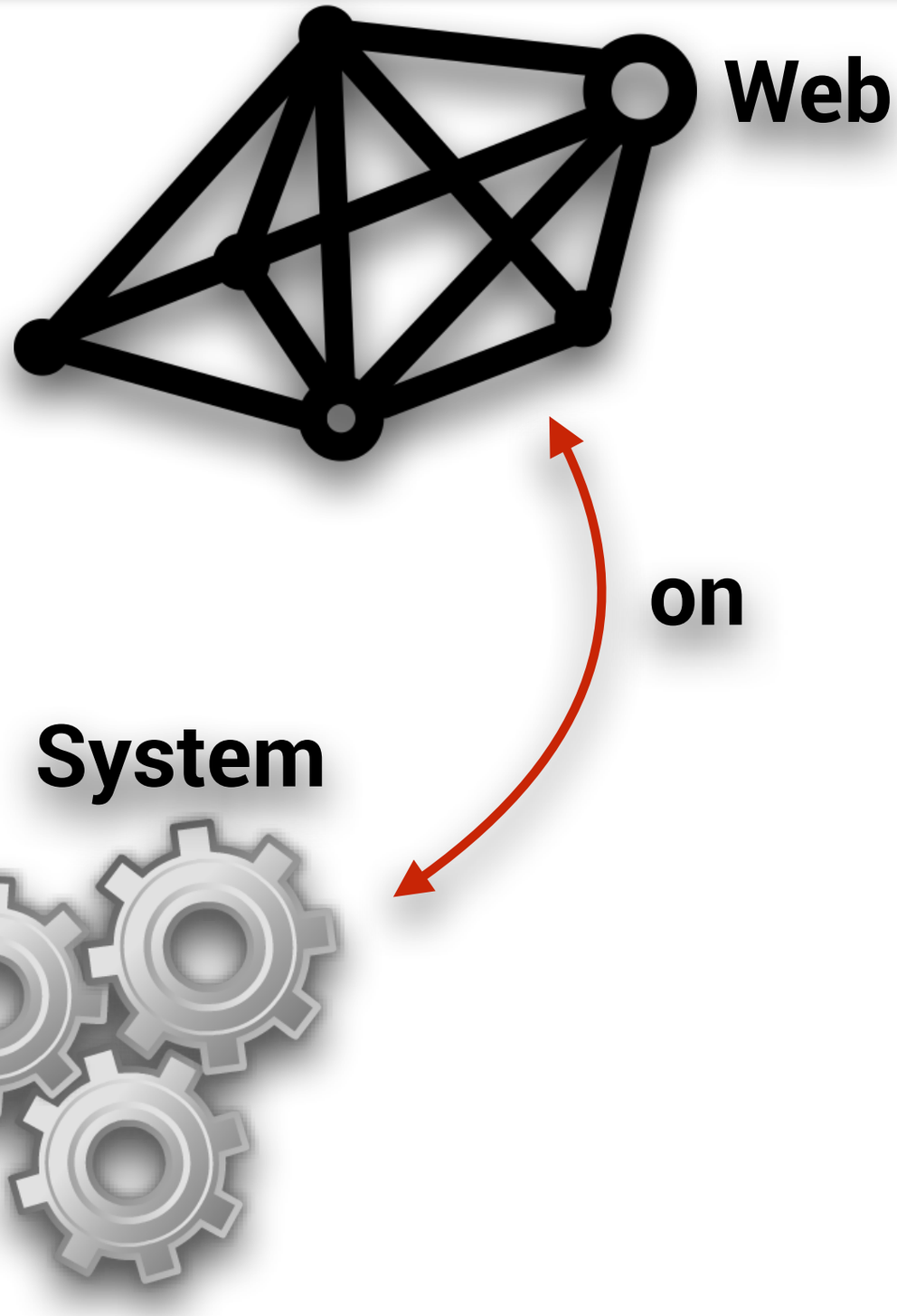
Data from/for the Web

- Data items that are
  - **generated by processes internal or external to** the Web
  - captured **and** stored
  - served out in response to **query** requests



Data on the Web

- Data items that are
  - **available on** the Web
  - captured but **not necessarily** stored
  - served out in response to **web-retrieval** requests



MANCHESTER 1824

The University of Manchester

A Web of Data

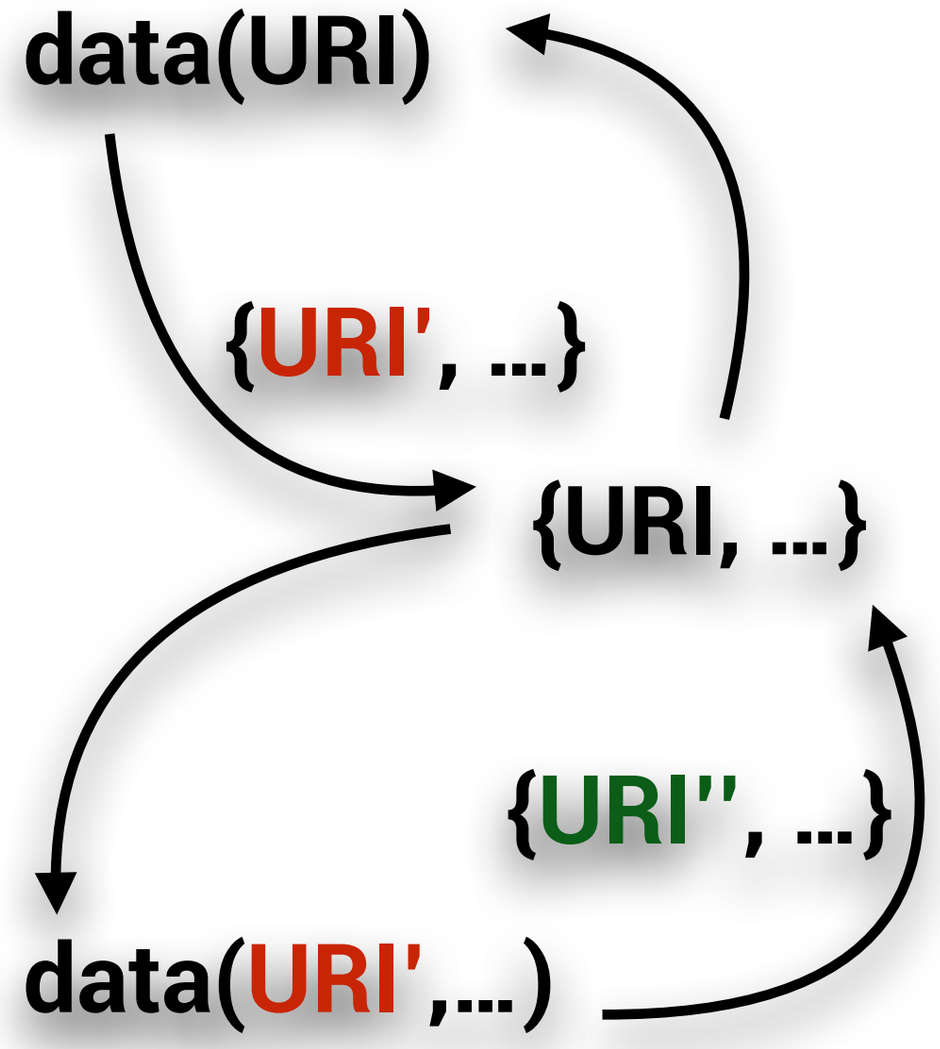
Web of Data: Linked Data Principles

1. Use URIs to name (identify) things.

2. Use HTTP URIs so that these things can be looked up (interpreted, "dereferenced").

3. Provide useful information about what a name identifies when it's looked up, using open standards such as RDF, SPARQL, etc.

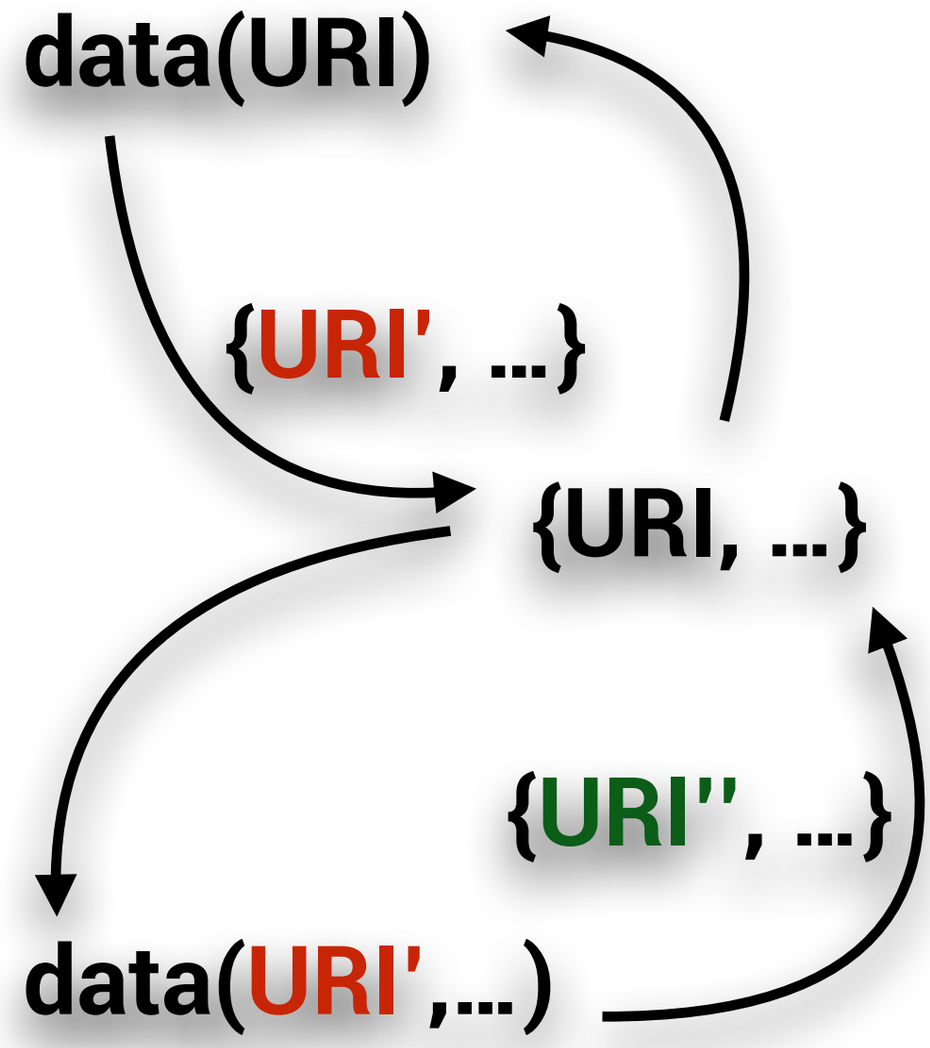
4. Refer to other things using their HTTP URI-based names when publishing data on the Web.



(Bizer, Heath, Berners-Lee, 2009)

Web of Data: Problems

1. Where to start?
2. How to request what you want?
3. How to make sense of the responses?
4. Where to link to next?
5. When to stop?



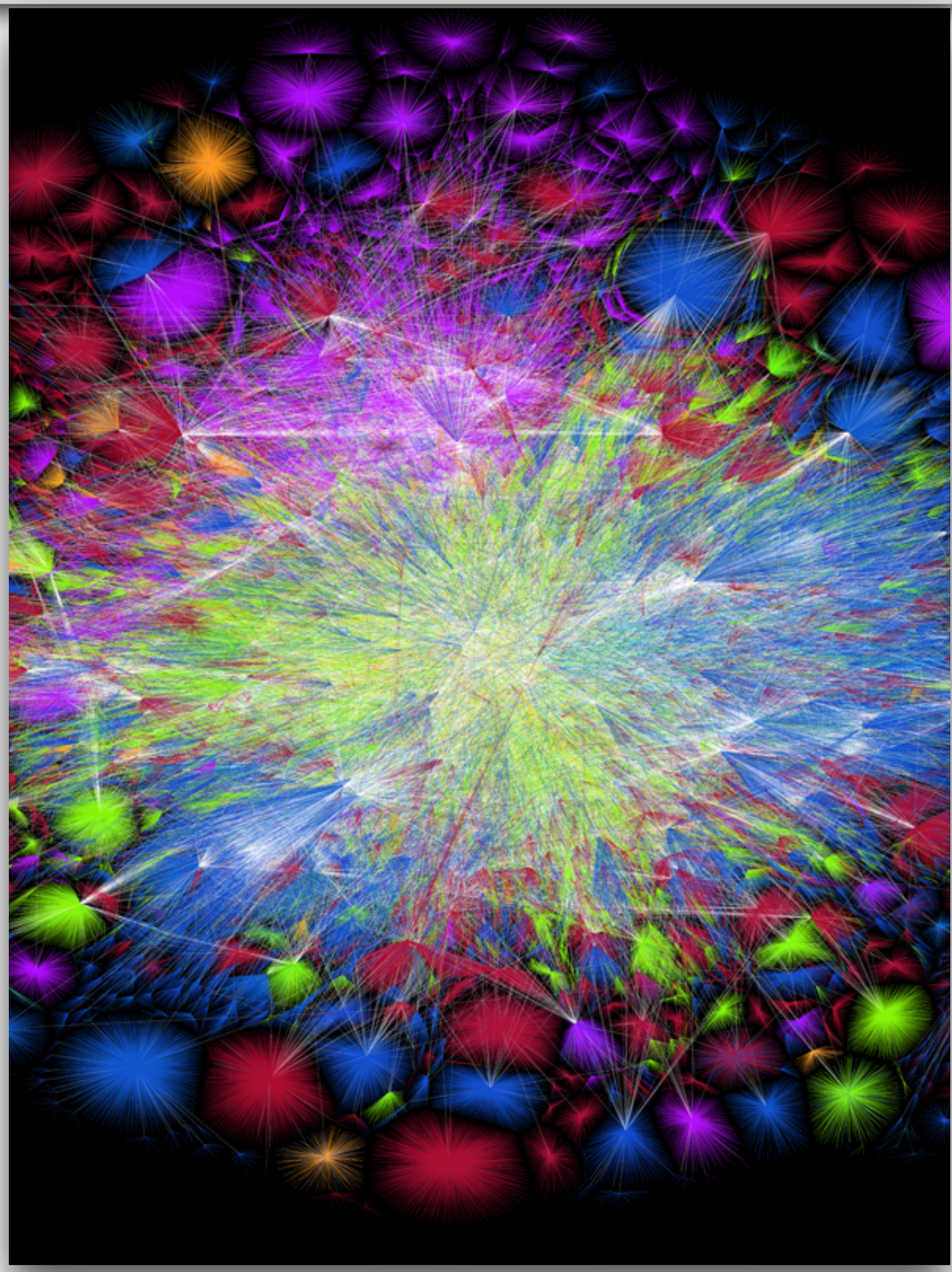
MANCHESTER 1824

The University of Manchester

The Wild Wild Web

The Web

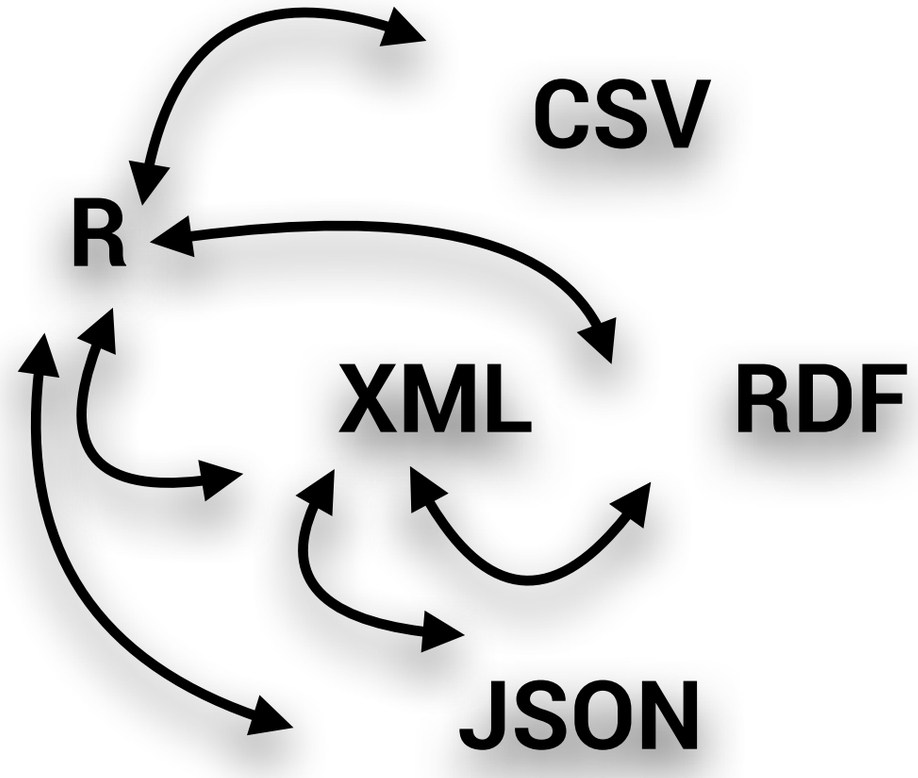
- Data **on** the Web is
  - massively distributed
  - extremely volatile
  - intractably voluminous
  - continuously, rapidly growing
  - paralysingly heterogeneous



The Internet on 11th July 2015 [opte.org]

Heterogeneity

- Syntactic heterogeneity is not so hard to reconcile.
- We have seen that relational, XML/JSON and RDF data can, with minor limitations, be made to be interchangeable and interoperable.



Heterogeneity

- Semantic heterogeneity is extremely hard to reconcile.
- Given a source S and a target T:
  - Price? With sales tax? Without?
  - Address? Simple? Composite?
  - Employees? Only part-time? Only full-time? Both?

$Price_T$? $Price_S$? $SalesTax_S$

$Address_T$? $HouseNos$? $Postcodes_S$)

Emp_T ? PTs ? FTs

Heterogeneity

- We need to identify concept equivalence then induce (or define) a mapping (typically a view) relating source and target.
- For example, in the target, Employee becomes the union of Part-Time and Fill-time in the source.

$Price_T \leftarrow Price_S + SalesTax_S$

$Address_T \leftarrow HouseNos_S ++ f(Postcodes_S)$

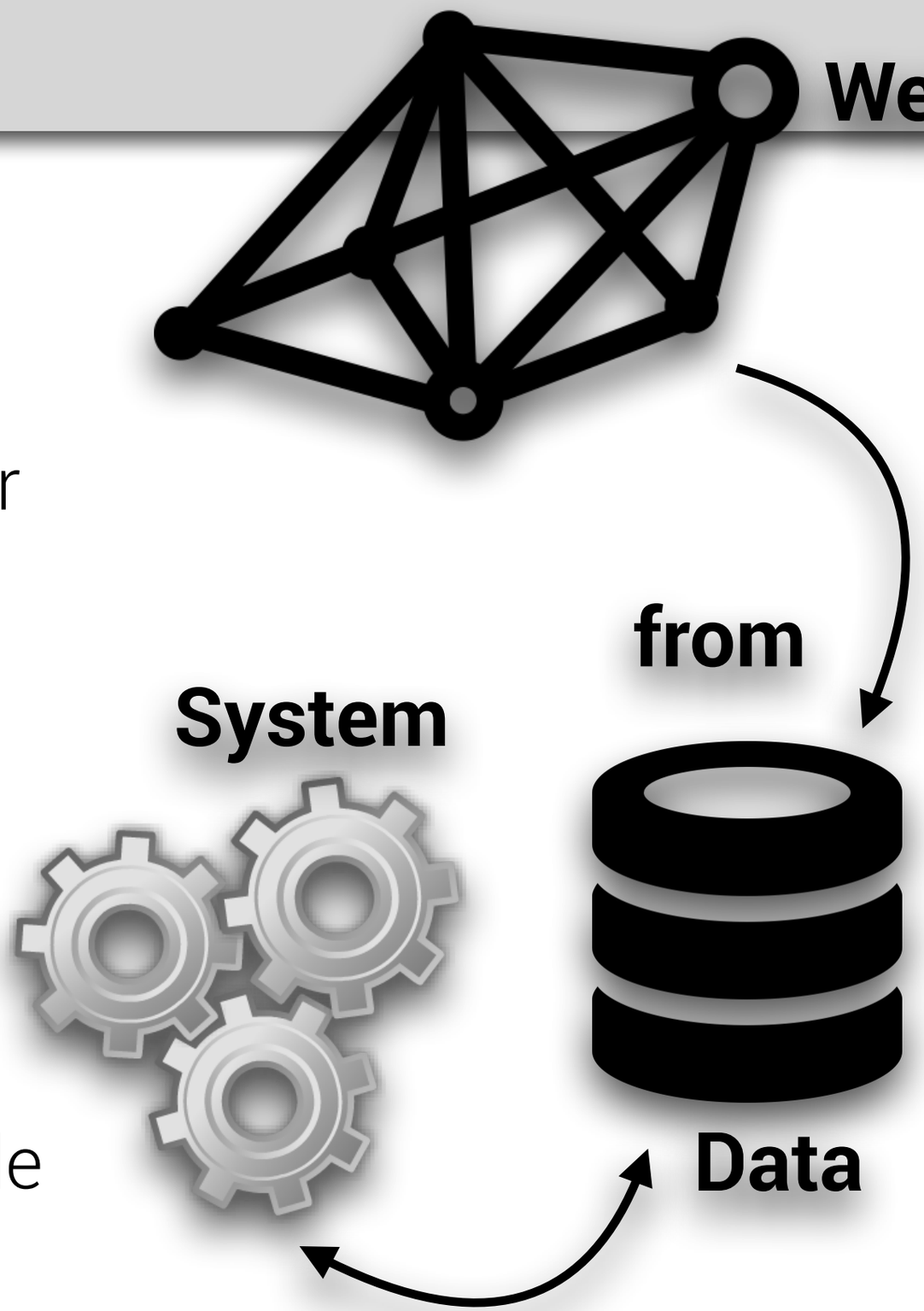
$Emp_T \leftarrow PT_S \cup FT_S$

The University of Manchester

Data on the Web of Data

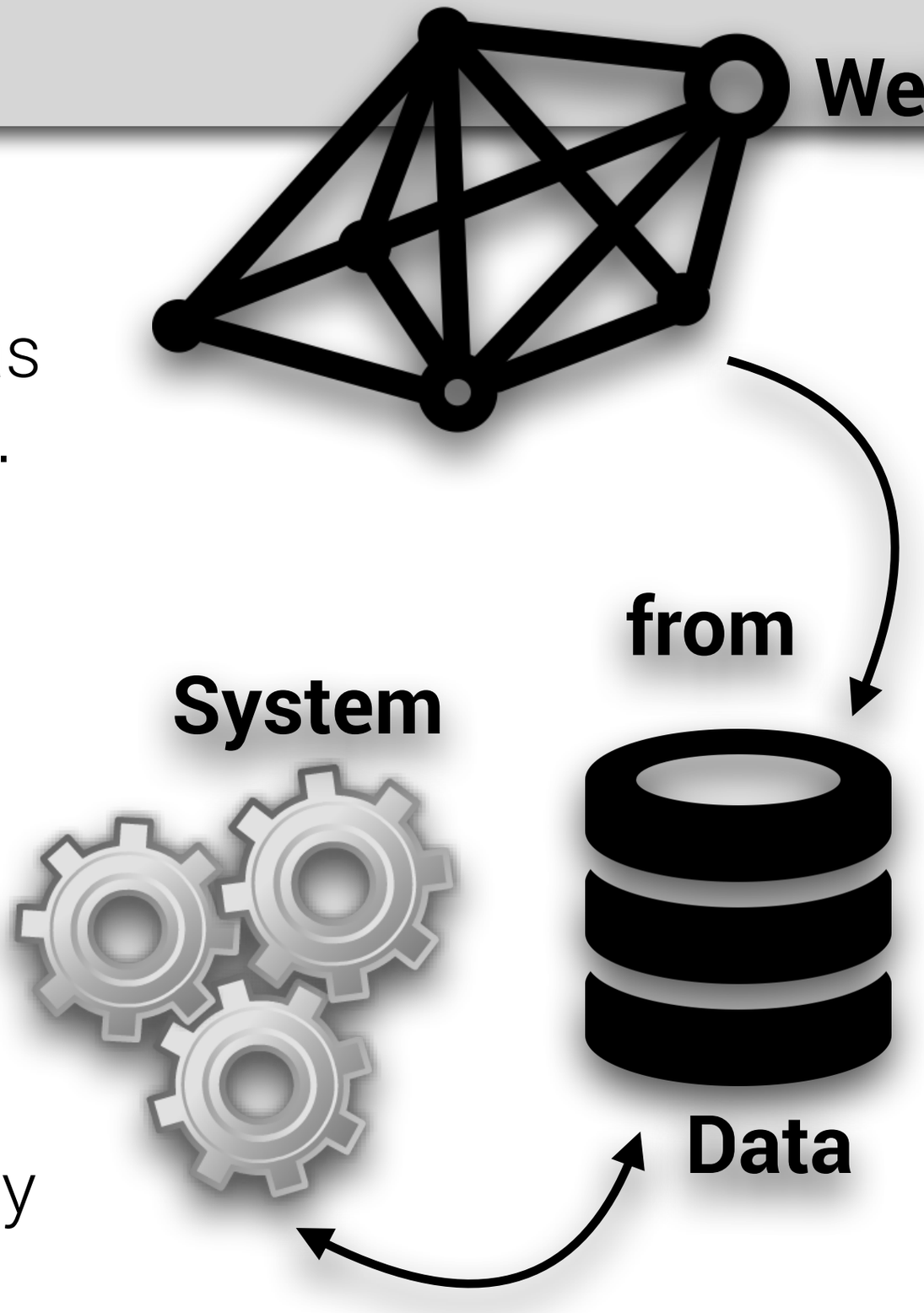
WoD by Warehousing

- Crawl to construct (or else fetch) a dump of a subgraph.
- Hold on to it for processing or pour it into a local SPARQL endpoint for querying.
- This does address the distribution issue, but not volume, volatility and growth.
- It remains a problem to decide where to start and when to stop.

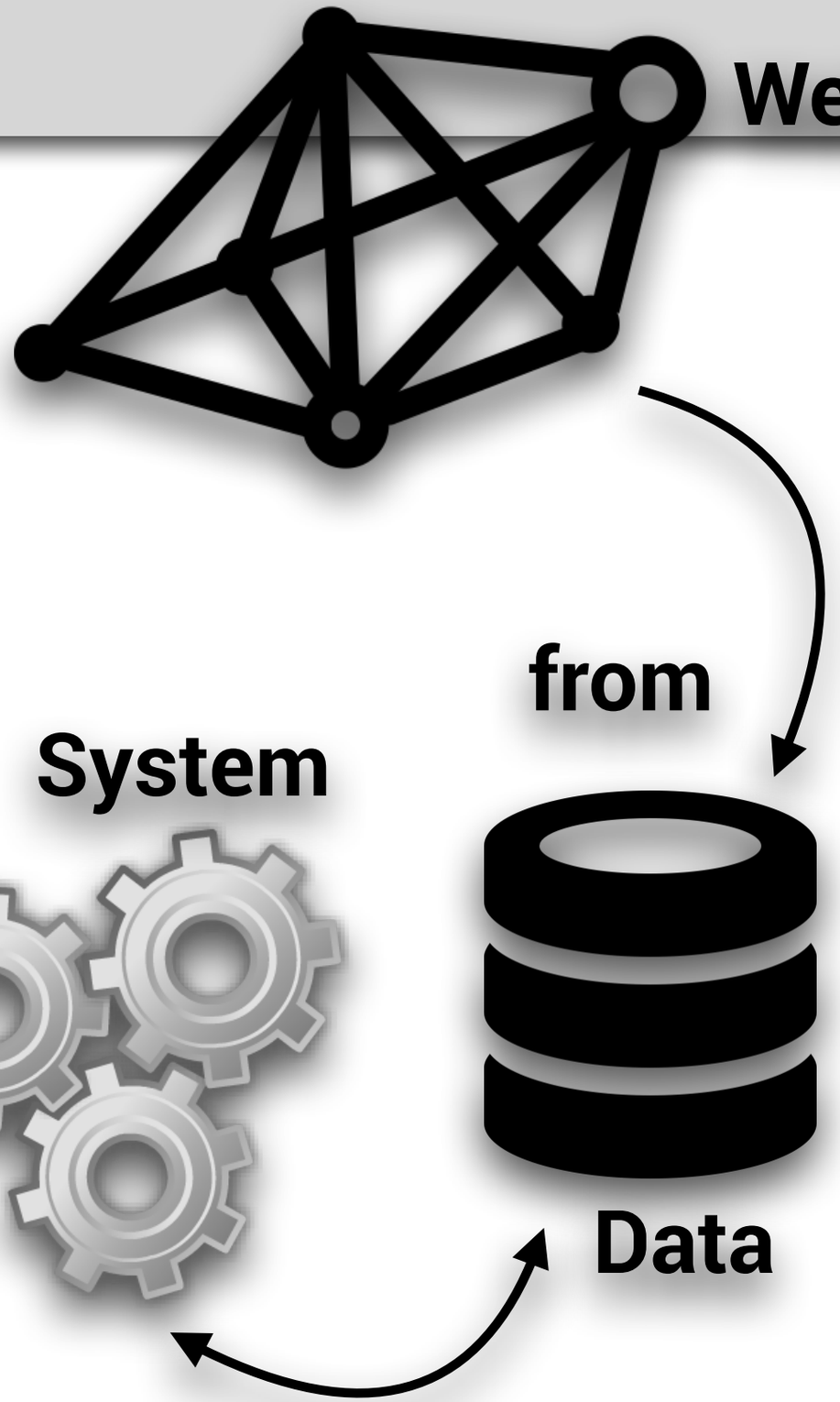


WoD by Warehousing

- Warehousing does not guarantee good recall, even as precision degrades over time.
- This is because synchronization/refresh is infeasible.
- If so, veracity/recency is compromised.
- The approach becomes costly while yielding limited value.



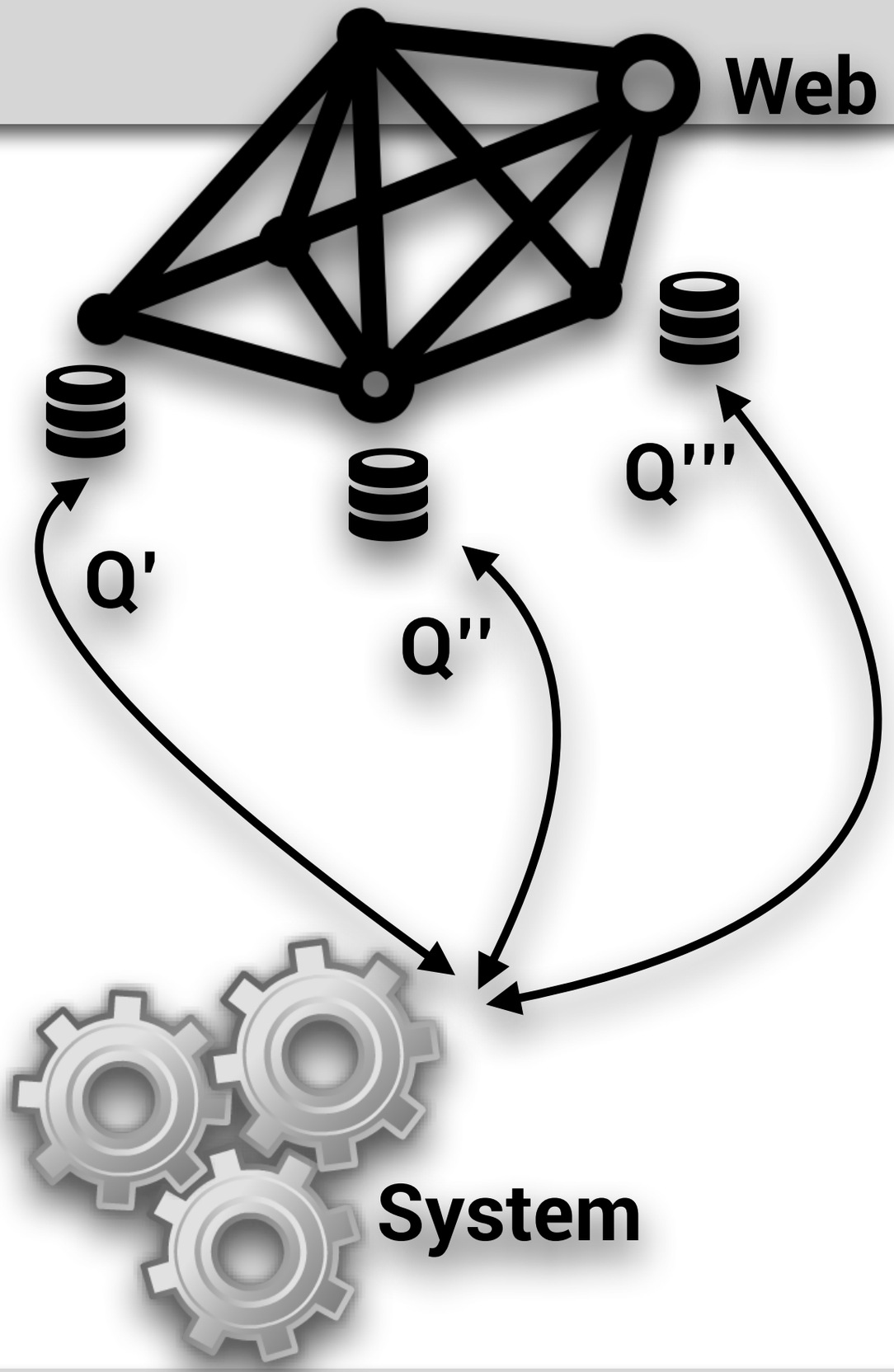
WoD by Warehousing



- The major benefits are response time and availability.
- Also, with locality, it is probably easier to address the simpler aspects of semantic conflict reconciliation.

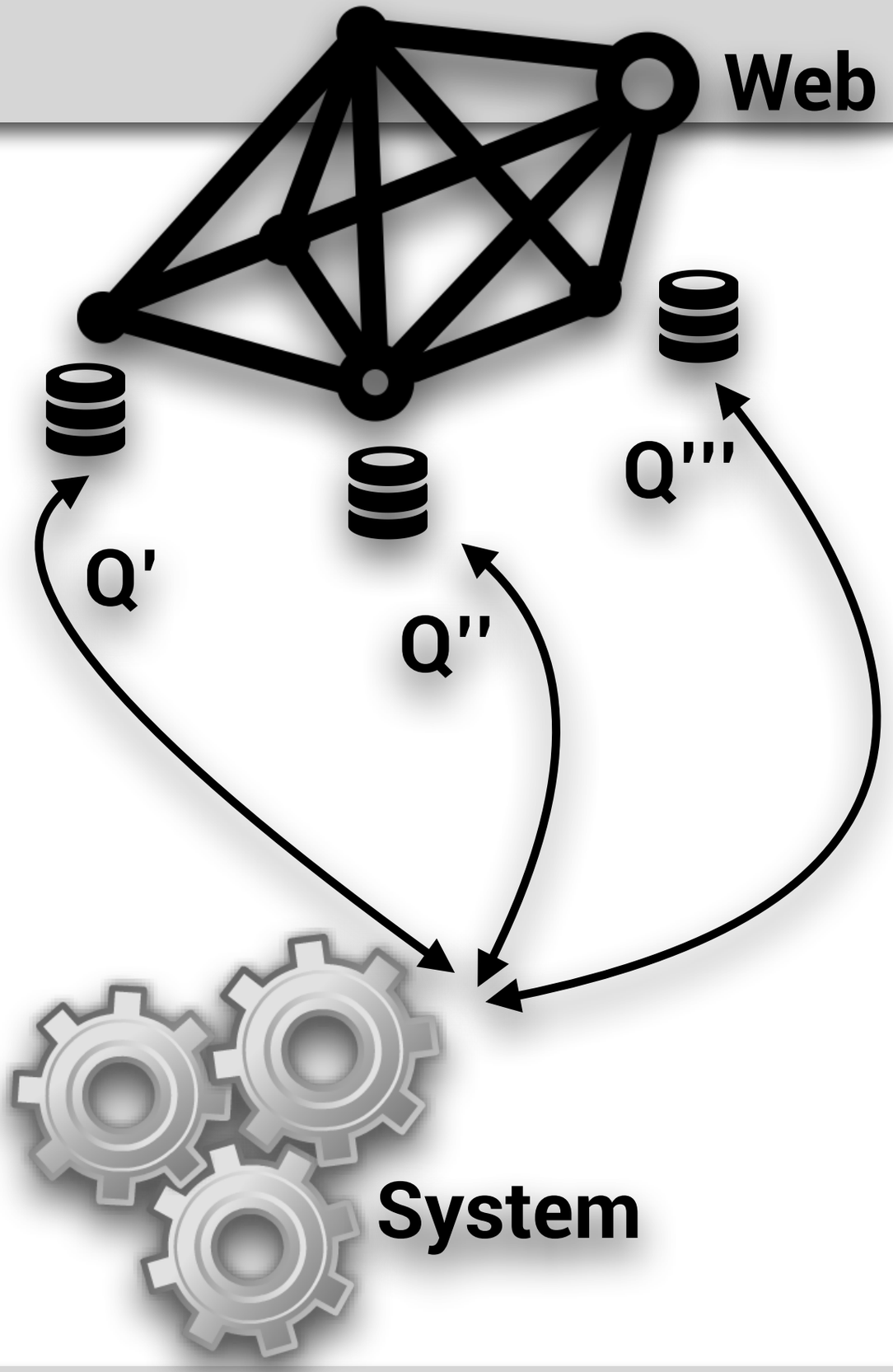
WoD by Federation

- Another approach is to rely on the sources being remote SPARQL endpoints.
- We can then issue SPARQL queries to request the data on the web we want.
- This is not so much a Web of Data but a Web of Databases.

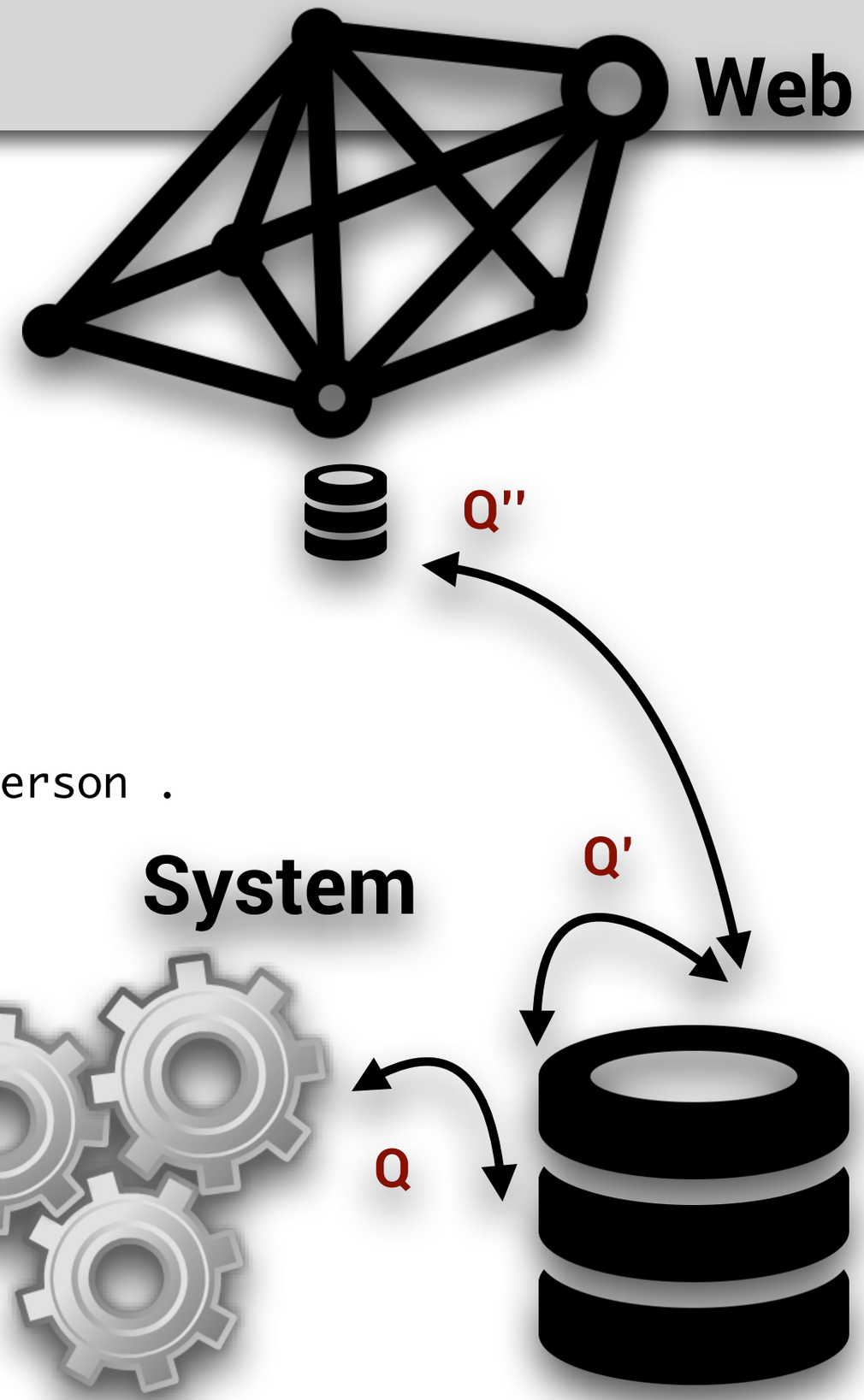


WoD by Federation

- Data is current, so veracity/ recency is not compromised.
- Data is not replicated/ materialized so synchronization/refresh is not necessary.
- It is supported in SPARQL 1.1



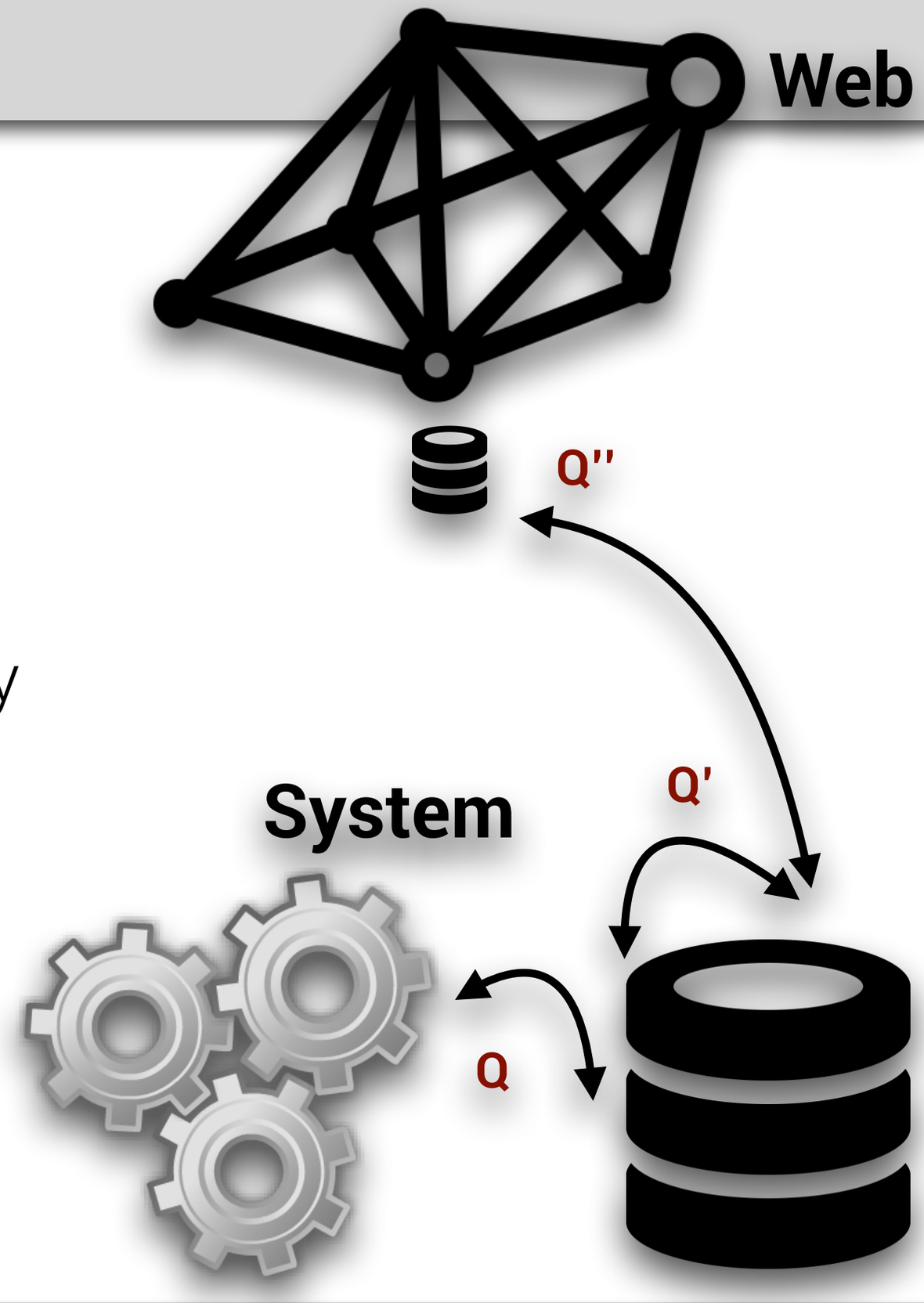
WoD by Federation




```
Q  PREFIX foaf:<http://xmlns.com/foaf/0.1/>
   SELECT ?name
   FROM   <http://example.org/myfoaf.rdf>
   WHERE
Q'{ <http://example.org/myfoaf/I> foaf:knows ?person .
      SERVICE <http://people.example.org/sparql>
  Q''{?person foaf:name ?name .
      }
   }
```

WoD by Federation

- Since some data items aren't held in databases, recall is compromised.
- Response time and availability cannot be guaranteed.
- Remoteness with autonomy makes reconciling semantic conflicts harder.



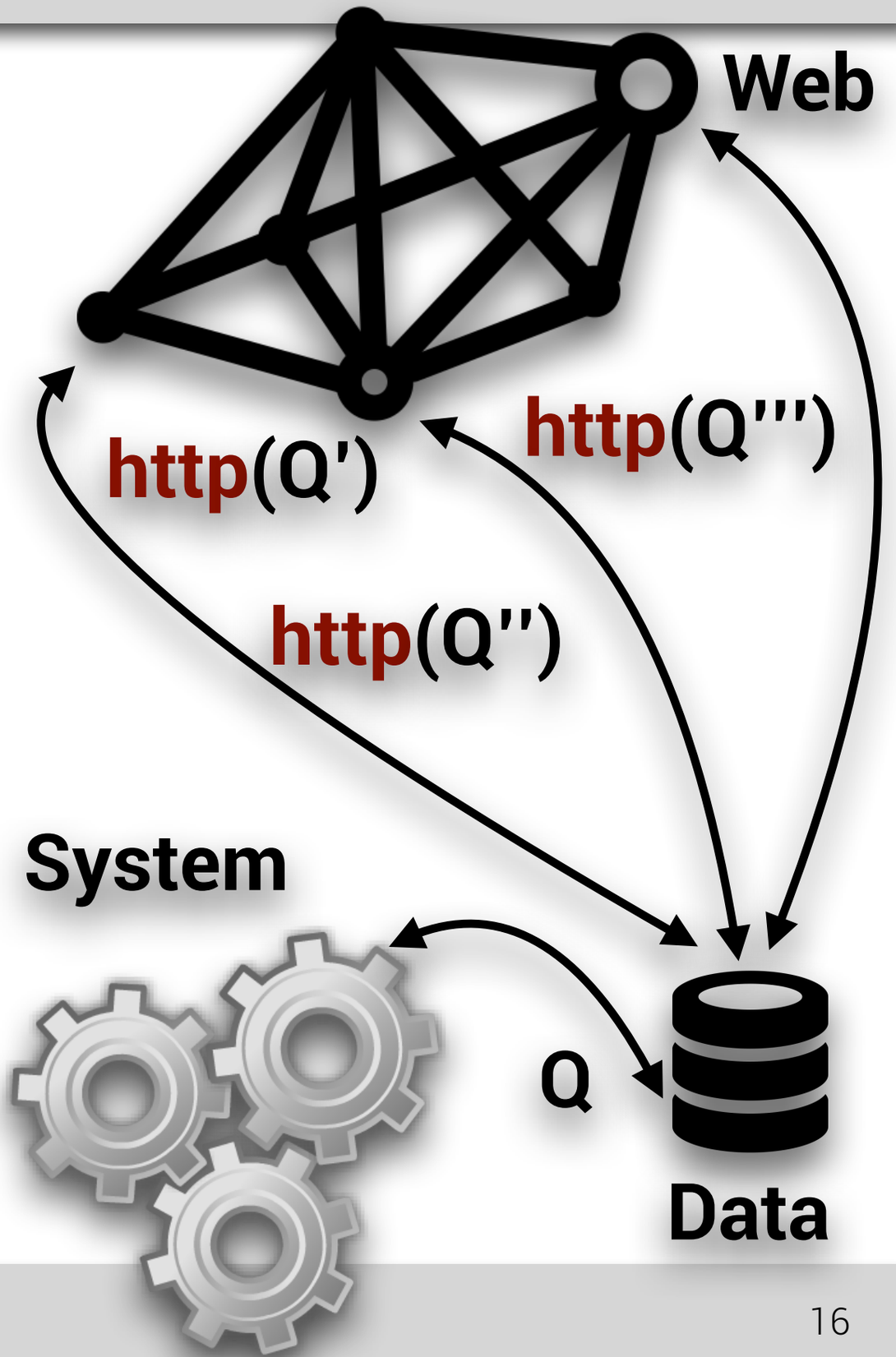
MANCHESTER 1824

The University of Manchester

The Web of (Linked) Data

Web of (Linked) Data

- This approach builds more strictly on the linked-data principles.
- Query evaluation is on live data on the Web.
- Potentially, <u>all</u> of the Web.
- A query evaluates against data that must be obtained, **on-the-fly, on the Web**.



Approaches to Problems

1. **Where to start?**
2. **How to request what you want?**
3. **How to make sense of the responses?**
4. **Where to link to next?**
5. **When to stop?**

Source Ranking

Source Selection

HTTP!

VERY HARD!

HTTP!

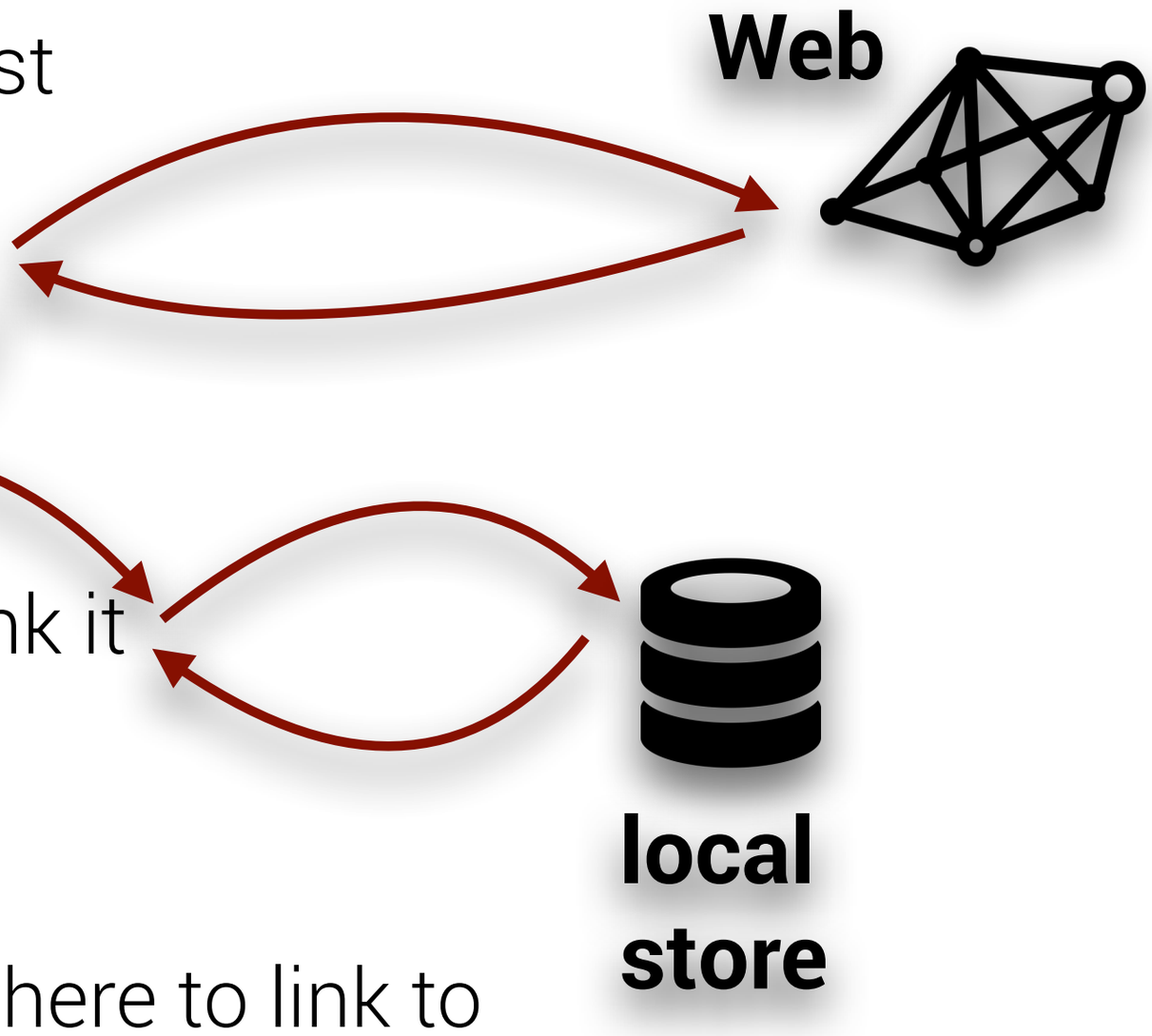
"Follow your nose?"

Result Ranking

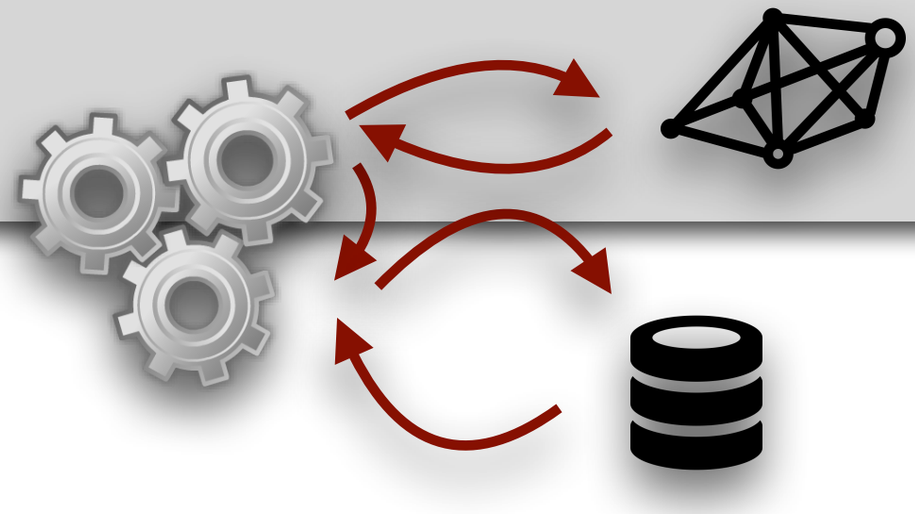
Result Construction

Solution Sketch

1. Select source to go to first
2. Request what you want
3. Cache the response
4. Update the result and rank it
5. Is it as required?
    1. No, therefore decide where to link to next and go to (2)
    2. Yes, therefore emit the result

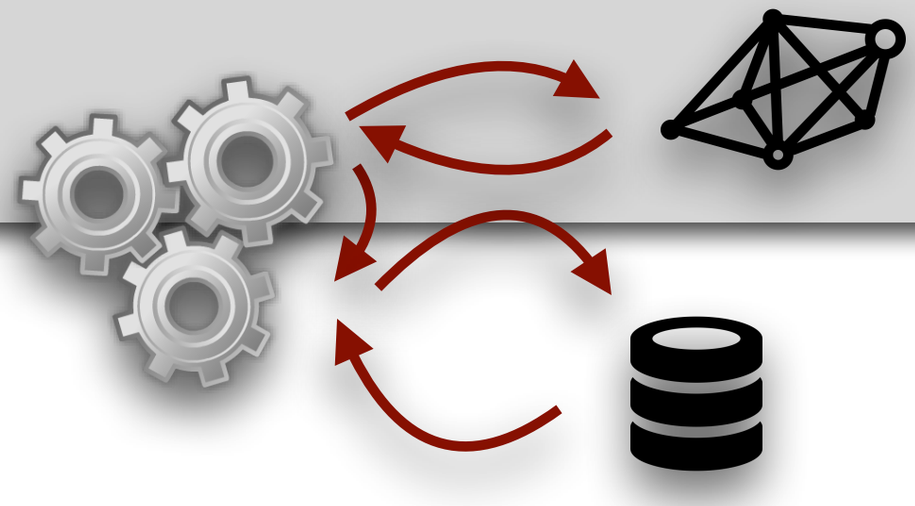


Advantages



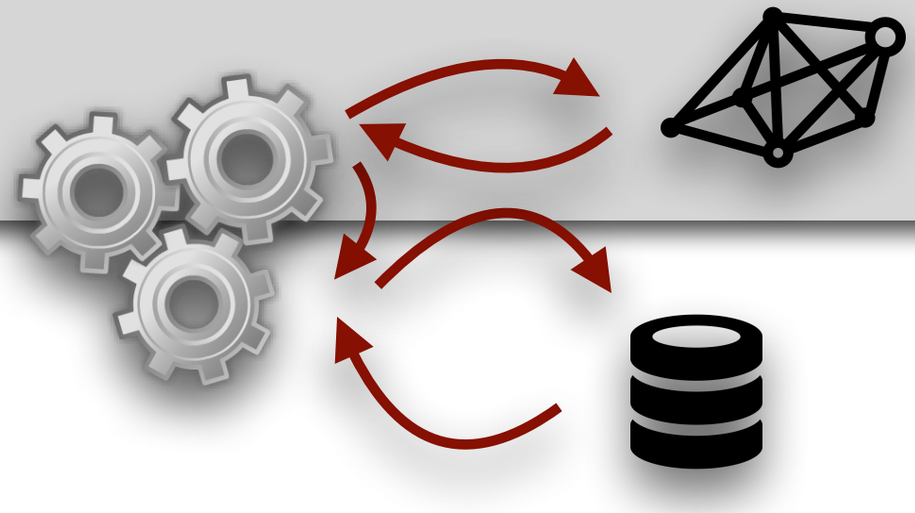
- Data is current, so veracity/recency is not compromised.
- Data is not replicated/materialized so synchronization/ refresh is not necessary.
- The local store may cache, but does not, strictly, materialize.
- Any, if not all the, data on the Web can be made available.

Advantages



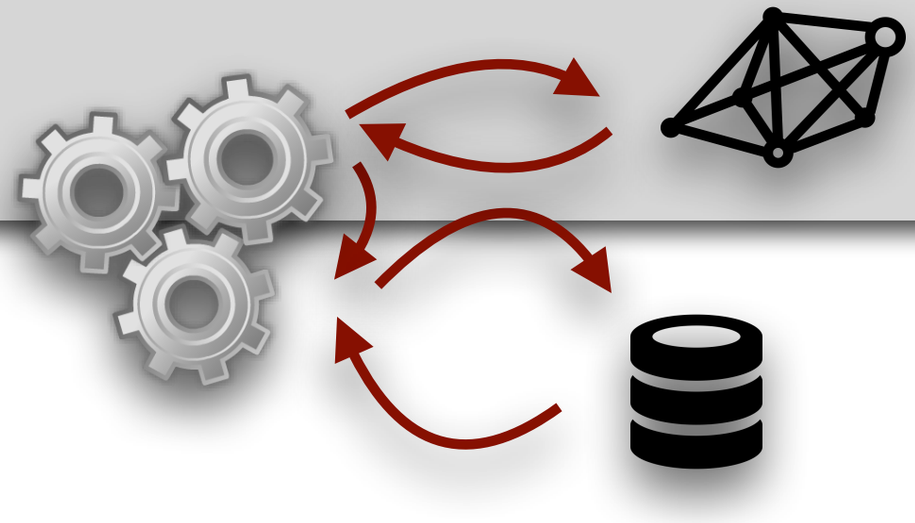
- It is a good match with SPARQL 1.1 capabilities, i.e., it does abide by the linked data principles.
- For example, retrievals from the Web can interleave with requests to SPARQL endpoints using **`DESCRIBE`**, **`ASK`**, **`SERVICE`** and **`CONSTRUCT`** queries.
- The SPARQL protocols unify the interaction with the Web and the local store (including **`NAMED GRAPH`** queries).

Drawbacks



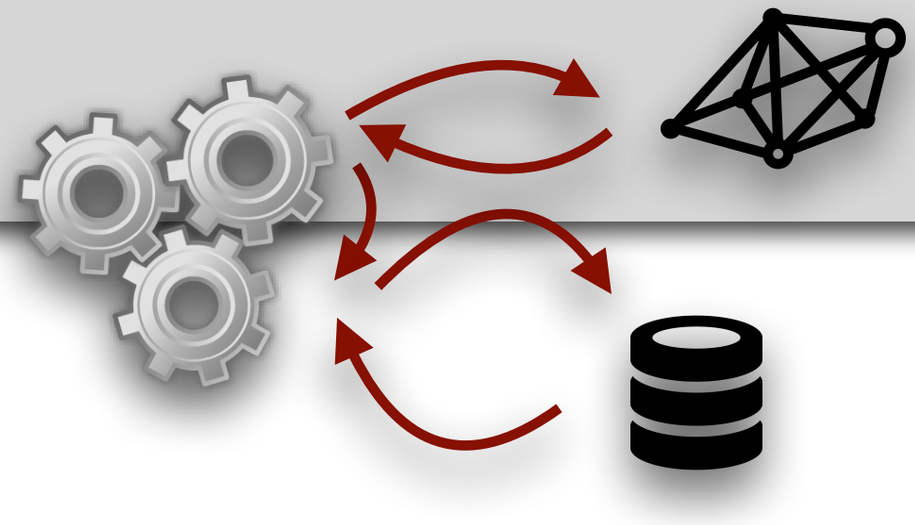
- In distributed query processing, data transfer over the network is many orders of magnitude more costly than even secondary memory access.
- Sending the query over to where the data lies is almost always more efficient than shipping the data to where the query is being issued.

Drawbacks



- The federated approach only ships data that the root of a query plan emits.
- This, by definition, has greater value (because it is more refined) than the leaf-level data.
- The linked data approach ships leaf-level data to where the query is being processed, which is the costliest strategy, in principle.

Drawbacks



- The federated approach is a better context for resolving heterogeneity, because SPARQL (like XQuery and SQL) can perform semantic reconciliation over remote sources.
- The pure linked data query processing approach would need to layer semantic reconciliation over live-data query execution.

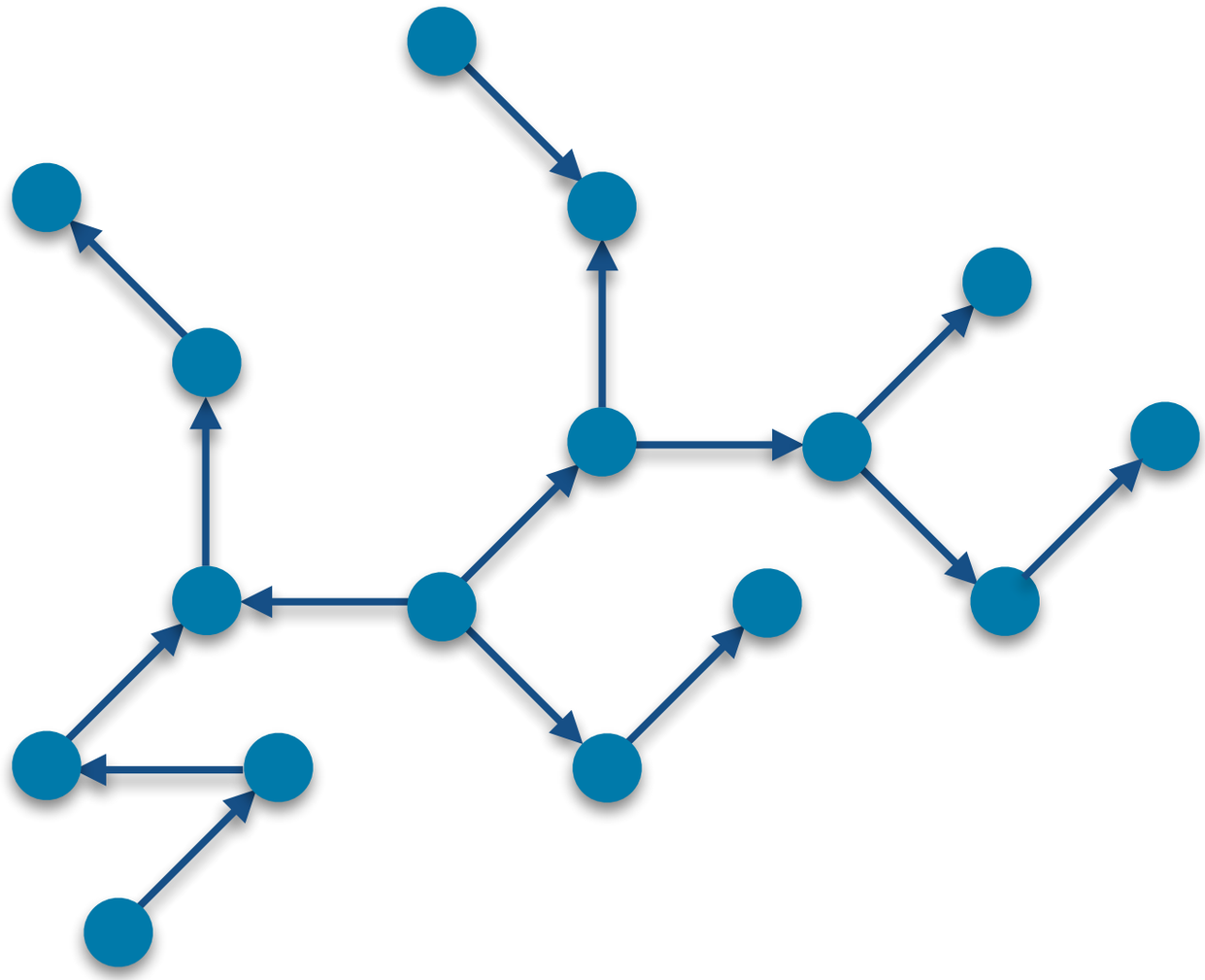
MANCHESTER 1824

The University of Manchester

Link-Based Query Evaluation

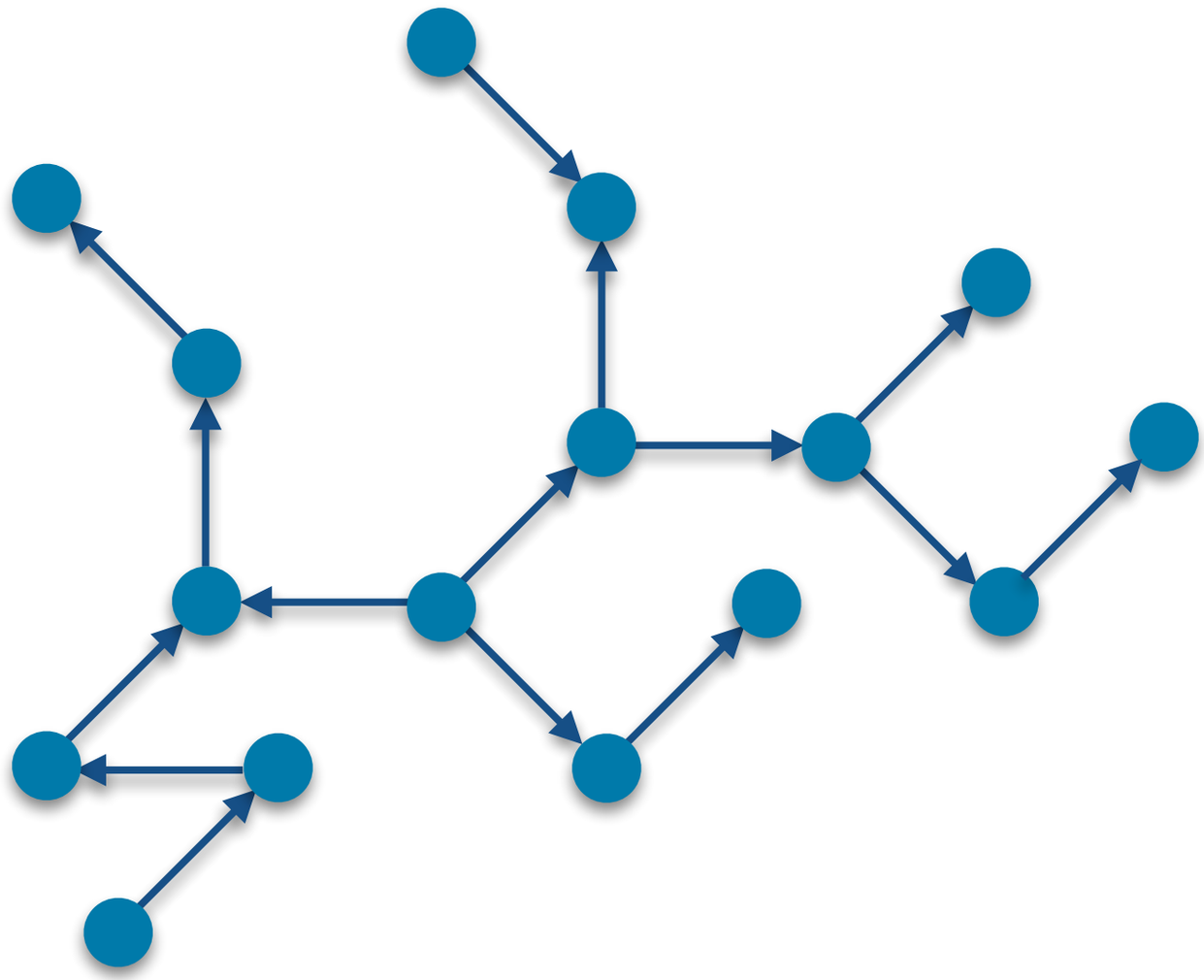
Link-Based Query Evaluation: Reachability-Based Semantics

- Given the linked data graph, we need to answer the questions we posed before:
  - Where do we start?
  - Where do we link to next?
  - Where do I stop?



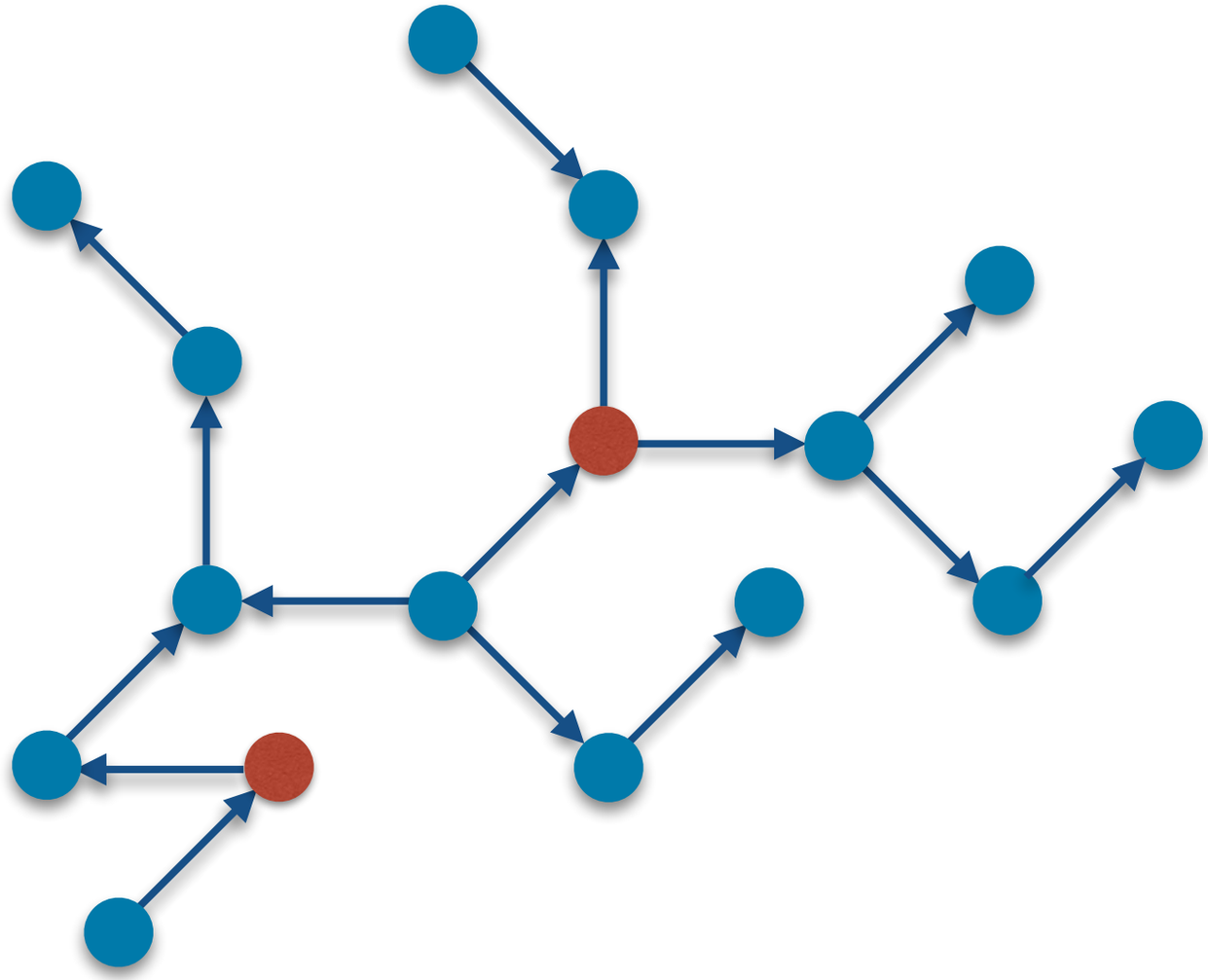
Link-Based Query Evaluation: Reachability-Based Semantics

- We must have answers so that the dataset over which the query will evaluate is precisely defined.
- Recall (from last week) that without a dataset the semantics of SPARQL evaluation is not well-defined.
- We define the dataset using a reachability criterion over the directed graph induced by the links in the data.



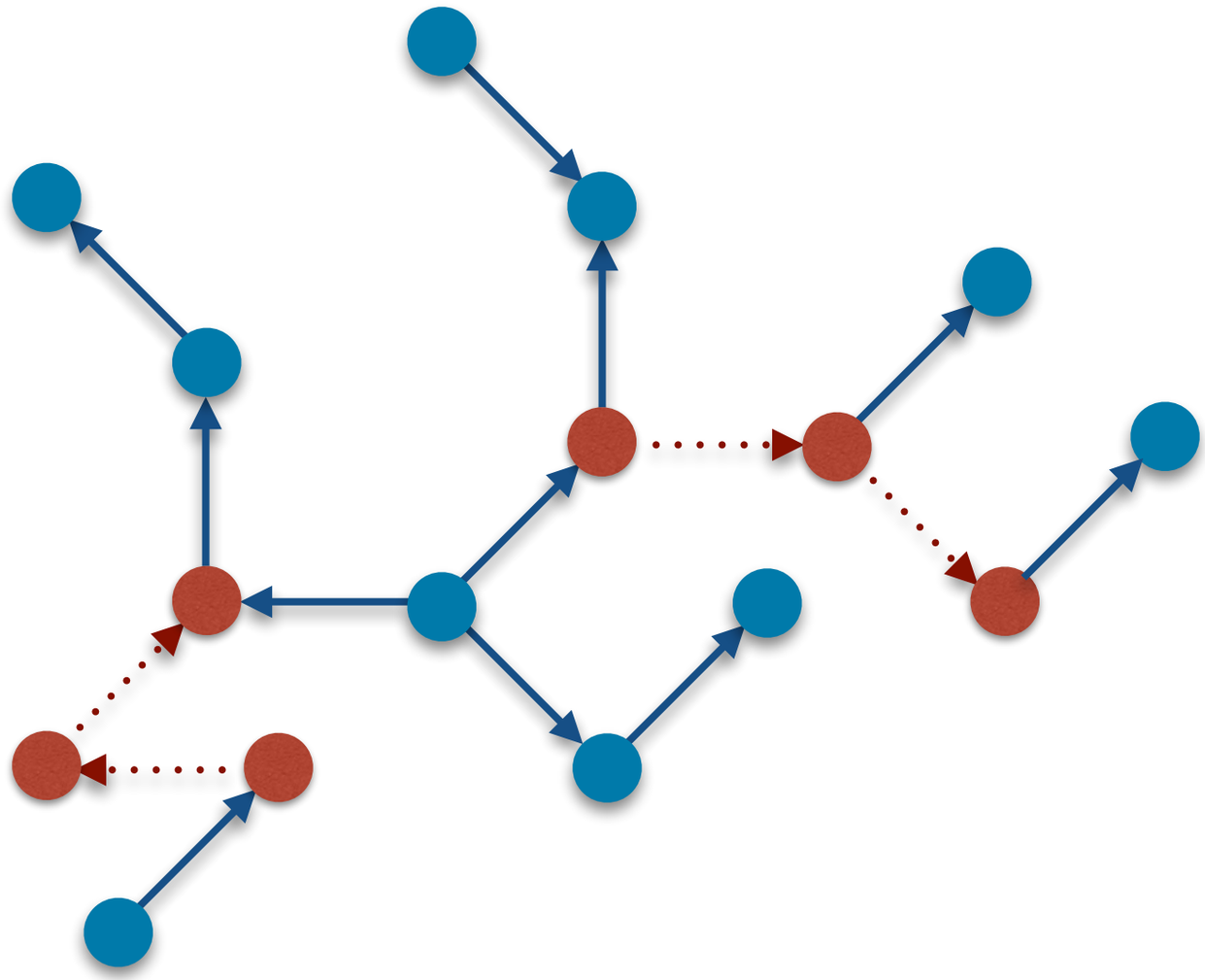
Link-Based Query Evaluation: Reachability-Based Semantics

- First, where do we start?
  - We start on a set of **seeds**.
- How do we choose the seeds?
  - We use a search engine, or
  - We use an index we have built by crawling, or
  - We used good seeds we have cached.



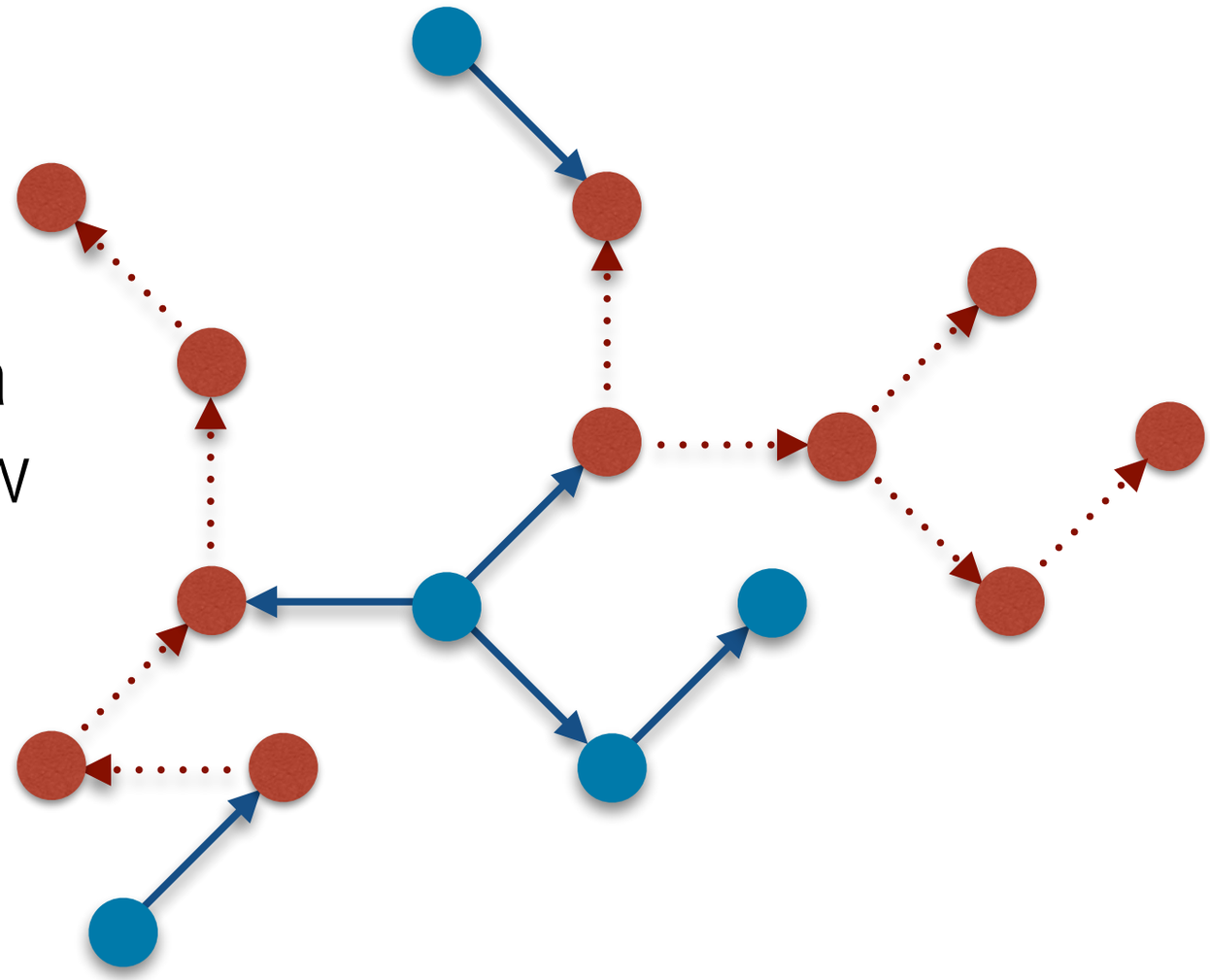
Link-Based Query Evaluation: Reachability-Based Semantics

- Second, where do we go next?
  - We "follow our nose", i.e., we dereference outgoing links.
  - This returns data that we may choose to add to our dataset in construction.



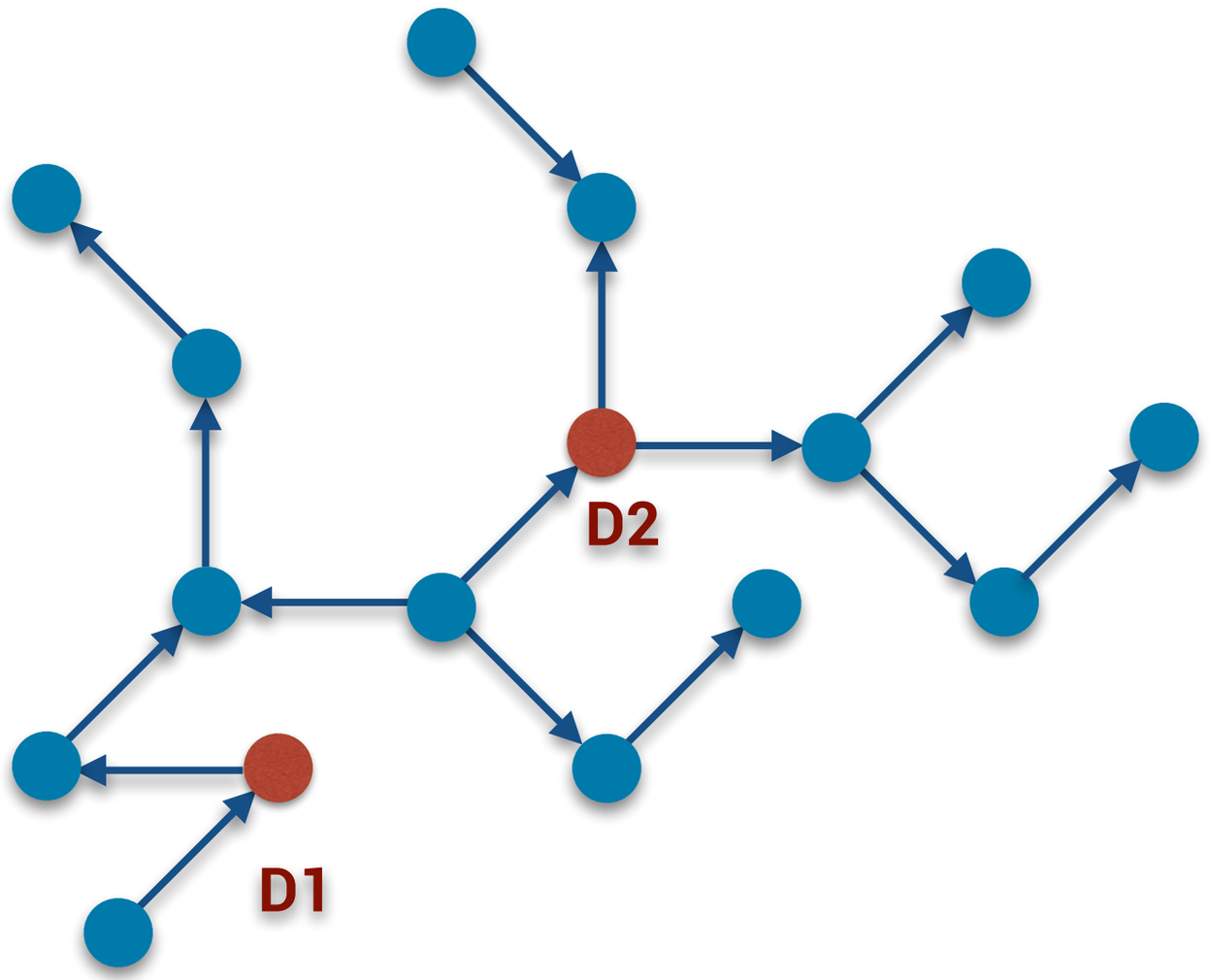
Link-Based Query Evaluation: Reachability-Based Semantics

- Third, where do we stop?
  - We stipulate a reachability criterion.
  - The reachability criterion is a decision on whether to follow a class of links or not.



Link-Based Query Evaluation: Reachability-Based Semantics

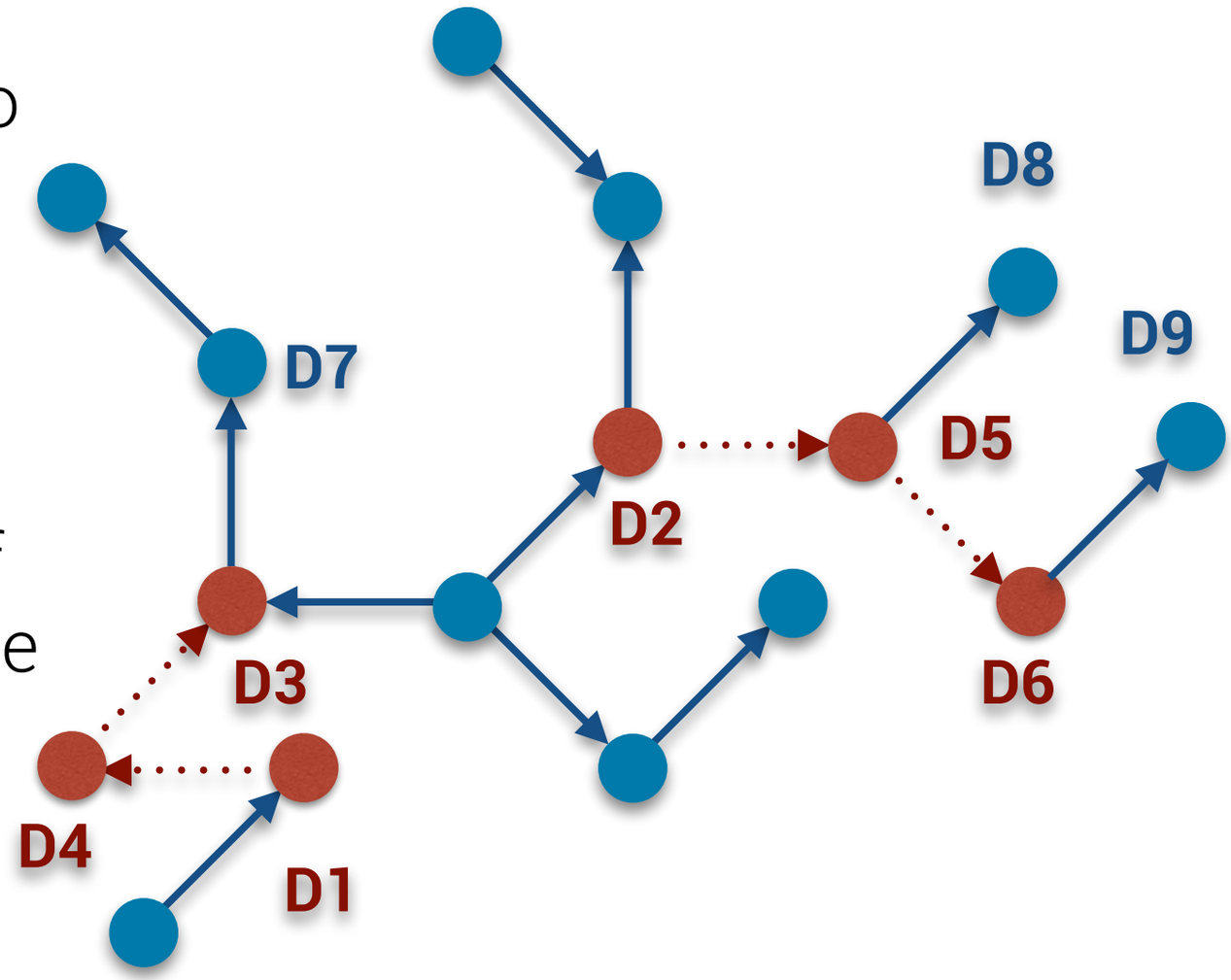
- One, minimalistic, approach is to stop at the seeds themselves, i.e., we do not traverse any links.
- The more relevant to a query our seeds can be estimated to be, the more efficient this approach is.
- Here, source selection/ranking takes the form of seed selection/ranking.



D = D1 ∪ D2

Link-Based Query Evaluation: Reachability-Based Semantics

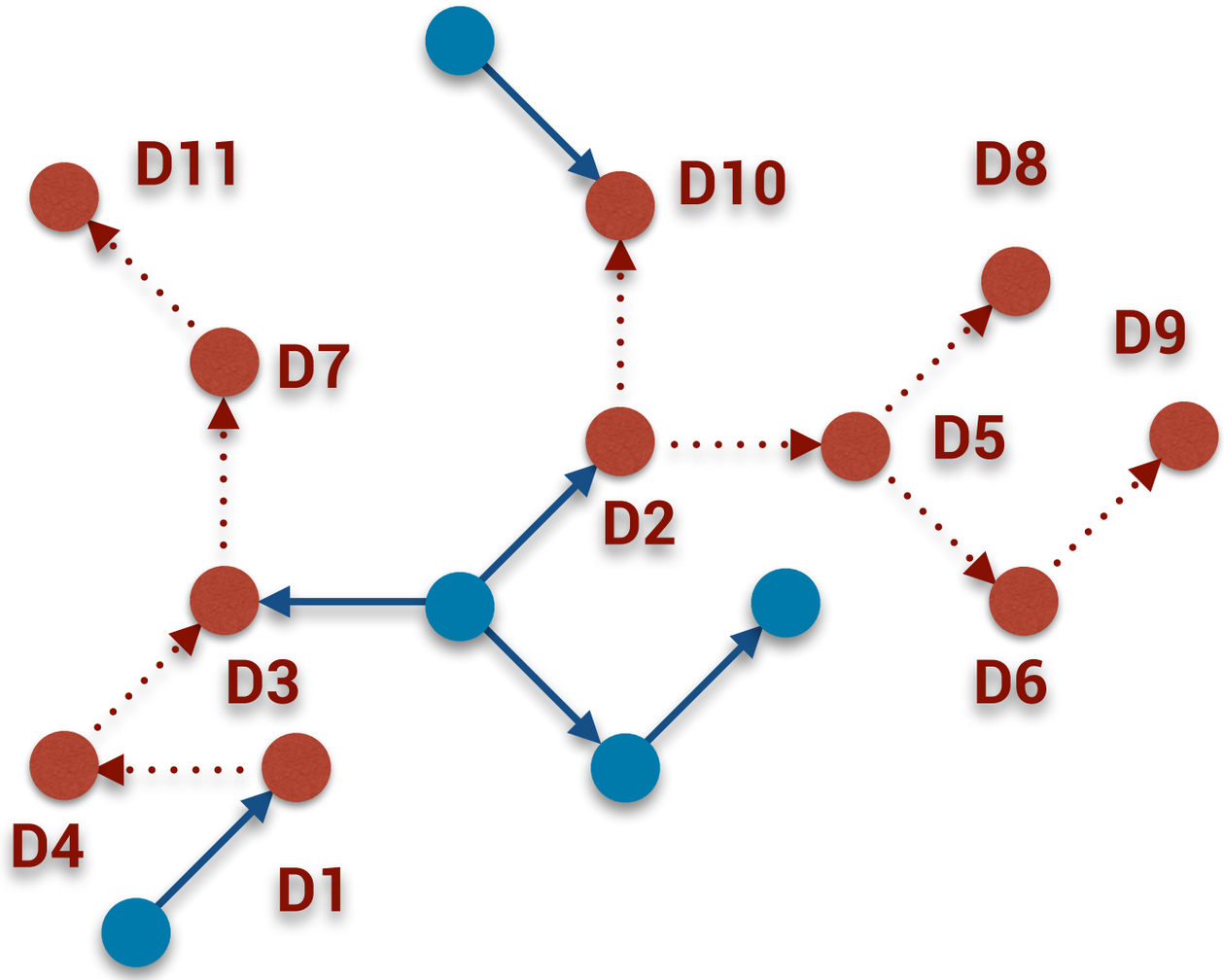
- Another, middle-ground, approach is to traverse while the links lead to matches.
- In the example,
  - we retrieve D1,
  - we dereference to retrieve D4 iff there are matches between triple patterns in the query and the triples in D4,
  - and again for D3,
  - but stop when D7 does not match.
- Correspondingly for the D2 seed.



D = (D1 ∪ D4 ∪ D3) ∪ (D2 ∪ D5 ∪ D6)

Link-Based Query Evaluation: Reachability-Based Semantics

- Another, maximalistic, approach is to traverse all the links.
- In the example,
  - we retrieve D1,
  - we dereference all the descendants of D1.
- Correspondingly for the D2 seed.



D = (D1 ∪ D4 ∪ D3 ∪ D7 ∪ D11) ∪
(D2 ∪ D5 ∪ D6 ∪ D8 ∪ D9 ∪ D10)

When is Link-Based Query Evaluation a Good Solution?

- The best scenarios for applying link-based query evaluation are characterized by the following criteria:
  - freshness of results is more important than response time
  - uncertainty (e.g., due to staleness of caches and indices) as to what data sources there are and what they contain needs reducing
- One concrete example is ad-hoc travel planning.
- Another is data journalism for fast developing news stories.

References

1. Bizer, Christian; Heath, Tom; Berners-Lee, Tim (2009). **Linked Data —The Story So Far**. International Journal on Semantic Web and Information Systems 5 (3): 1–22. http://dx.doi.org/10.4018/jswis.2009081901

2. Hartig, Olaf (2013). **Linked Data Query Processing**. Tutorial at the 22th International World Wide Web Conference (WWW), May 2013, Rio de Janeiro, Brazil. http://db.uwaterloo.ca/LDQTut2013/

Lecture 22

Massively-Parallel Schemes: NoSQL

Section 1

DBMSs, Then and Now

DBMSs, Then and Now (1)

What have classical DBMSs been good at?

- Classical DBMSs have been good when
  1. processing data that is structured in the form of records (hence with known, stable structure),
  2. only a few records are operated upon per task, e.g., in on-line transaction processing (OLTP),
  3. writes dominate performance (because of locking, latching, logging and buffer management),
  4. data growth can be addressed with vertical scaling,
  5. there is no need for dynamically responding in real-time to external events,
  6. there is no need for embedding in the physical world in which organizations exist.

DBMSs, Then and Now (2)

How are DBMSs evolving?

- "The Web changes everything."
- It is big time for big data!
- Most cutting-edge DBMS research and products are geared towards supporting one or more of:
  1. Semi- and unstructured data too,
  2. on-line analytical processing (OLAP), and also off-line, too,
  3. distributed, extremely voluminous data and computational resources with high horizontal scalability and high availability,
  4. dynamic response in real-time to external events,
  5. embedding in the physical world in which the organization exists.

Issues for Classical DBMSs (1)

Not Only Structured, Semi- and Unstructured Data Too

key:value stores : an influential exemplar is Amazon's Dynamo.

document stores : well-known exemplars are CouchDB, MongoDB, and Amazon's SimpleDB.

extensible record stores : influential exemplars are Google's BigTable and Facebook's Cassandra.

graph/triple stores : well-known exemplars are Neo4j (for general graphs), and Jena, Sesame, 4store (for RDF)

Issues for Classical DBMSs (1)

Not Only Structured, Semi- and Unstructured Data Too (1)

key:value stores :

- collections of objects,
- each with an application-assigned key
- tagging a schemaless payload
- with no DBMS-discernible structure;
- application program must do all the work
- using `put(key, value)`, `get(key):value` API.

Issues for Classical DBMSs (2)

Not Only Structured, Semi- and Unstructured Data Too (2)

document stores :

- collections of objects,
- each with an application-assigned key
- tagging a collection of attribute:value pairs;
- the DBMS knows about attributes and value types,
- there is a notion, therefore, of flexible, variable, possibly-nested schema structures;
- can be (non-SQL) queried (e.g., XQuery) or
- application program can use `set(key, document)`, `get(key):document`, `set(key, attribute, value)`, `get(key,attribute):value` API.

Issues for Classical DBMSs (3)

Not Only Structured, Semi- and Unstructured Data Too (3)

extensible record stores

- collections of key-sharing column families (like vertical partitions)
- where each column family is like a document (see above);
- no query-language support;
- application can use `define(family)`, `insert(family, key, columns)`, `get(family,key):  columns`

Issues for Classical DBMSs (4)

Not Only Structured, Semi- and Unstructured Data Too (4)

graph/triple stores
- collections of labeled edges over collections of labeled nodes
- with system-assigned IDs, and
- where both edges and nodes need not be predetermined, or fixed, or stable, and
- edge and node labels can be collections of attribute:value pairs;
- can be (non-SQL) queried (e.g., SPARQL) or
- application program can use `create: id`, `get(id):graphObject`, `connect(`$\mathtt{id}_1$`, `$\mathtt{id}_2$`):id`, `addAttribute(id,name,value)`, `getAttribute(id,name): value` API.

Issues for Classical DBMSs (5)

Not Only OLTP, OLAP Too

- While classical DBMSs do well on OLTP, they are less good at OLAP.
- OLTP typically consists of workloads that mix updates and queries whose profiles and demands can be studied and optimized for in advance.
- OLAP workloads are mostly read-only and dominated by ad-hoc (or transient) requirements.
- In modern organizations, the competitive edge that is provided by advanced data management is moving from transactions to analyses.

Issues for Classical DBMSs (6)

Not Only Scaled-Up Node Performance, Scale Out Too

- Classical DBMSs are not cheap to acquire, are costly to manage and tune, and costly to run.
- It has often been observed that the total cost of ownership (TCO) for DBMSs is increasing quite significantly.
- TCO = ACQ + ADM + OPE, i.e., TCO adds the acquisition cost, the costs of managing and maintaining the system (e.g., application development, database tuning) and the running costs (e.g., computational and storage resources, energy, etc.)
- Even if the acquisition component of TCO were not high, the administration and operation costs are high and rising.

Section 2

21st-Century: Big Data

Big Data (1)

What Do We Mean? The (Adapted) Gartner/Forrester Vs

Volume data in the order of terabytes per day, petabytes per year (close to 40% growth per year; 40x in the next 10 years): **scale-up fails, scale-out only option**

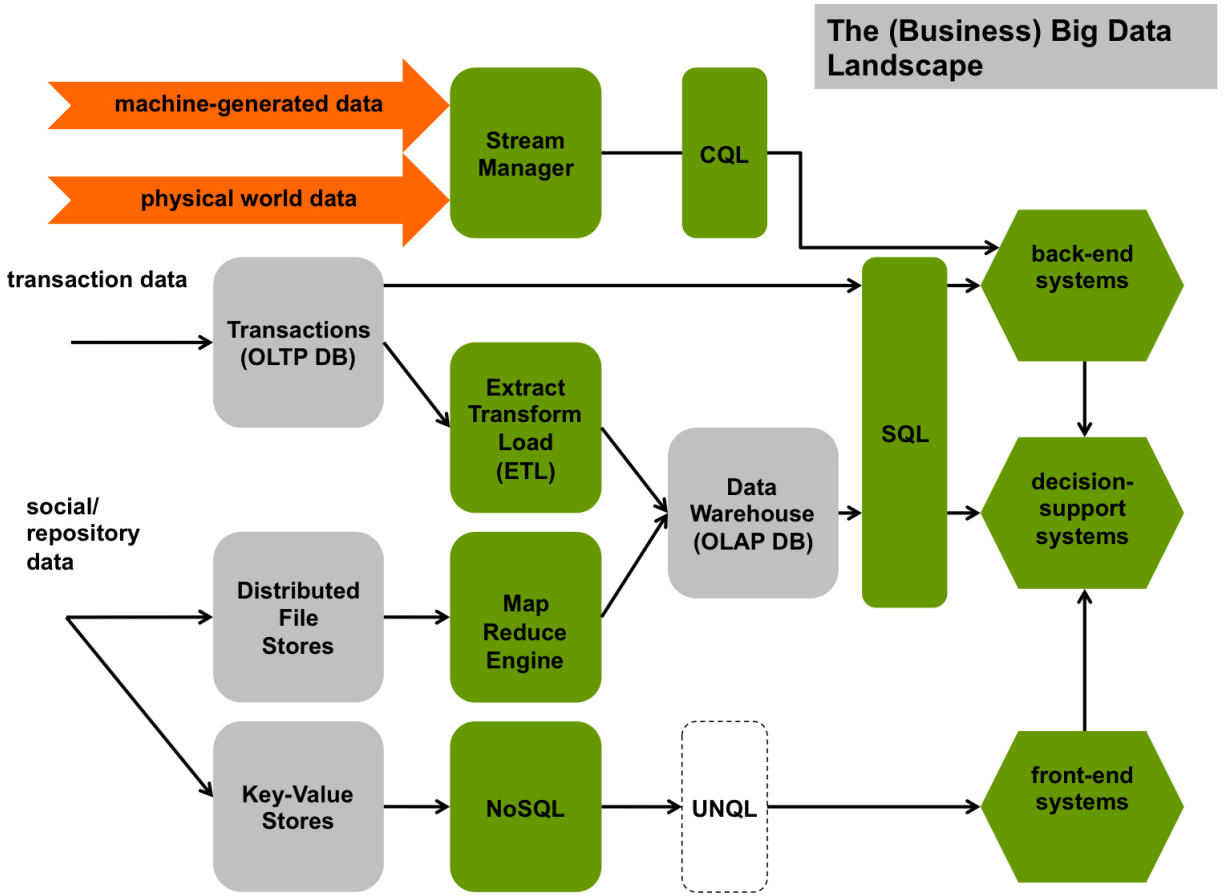
Velocity data that is retrieved statically or arrives continuously, on demand or not, non-stop, 24 x 7 x 4 x 12: **time to process too small compared to data change rate**

Variety data with no published stable semantics (e.g., documents, not just records; snippets, not often complete; text, often informal; media streams, often noisy; sometime human- and sometimes machine-generated): **impractical to integrate or evolve semantically**

Value in Variability data that underpins the construction of models (of businesses, markets, phenomena, behaviours) through the identification of what is regular/expected (or irregular/unexpected) in terms of threats and opportunities

Big Data (2)

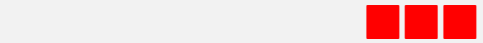
The Landscape [Werner Vogels's collect: store: organize: analyze: share (*)]



(*)
http://www.odbms.org/blog/2011/11/on-big-data-interview-with-dr-werner-vogels-cto-and-vp-of-amazon-com/

Big Data (3)

In Business Contexts (1)



traditional enterprise data

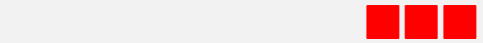
- customer/supplier relationship
- traditional transactional
- web store transctional
- general ledger (accounts, stocks, invoices, etc.)

social/repository data

- customer/market sentiment
- micro- and macro-blogging feeds
- social web
- documents, (image|video|sound) libraries/archives

Big Data (4)

In Business Contexts (2)



physical-world data

- sensing
- tagging
- positioning
- video/sound streams

machine-generated data

- call detail records, clickstreams, logs
- smart metering, equipment logs
- trading/transactional streams

Big Data (5)

Benefits in Business Contexts

- So much, so varied, so pervasive, so up-to-date data
- promises to enable a more thorough and insightful
- understanding of a business
- in itself and in relation to its customers, suppliers and competitors
- with a view to
  - enhancing productivity (e.g., eliminating inefficiencies in the value-adding chain)
  - improving scope to compete (e.g., on price, on return on investment, on quality of service, etc.)

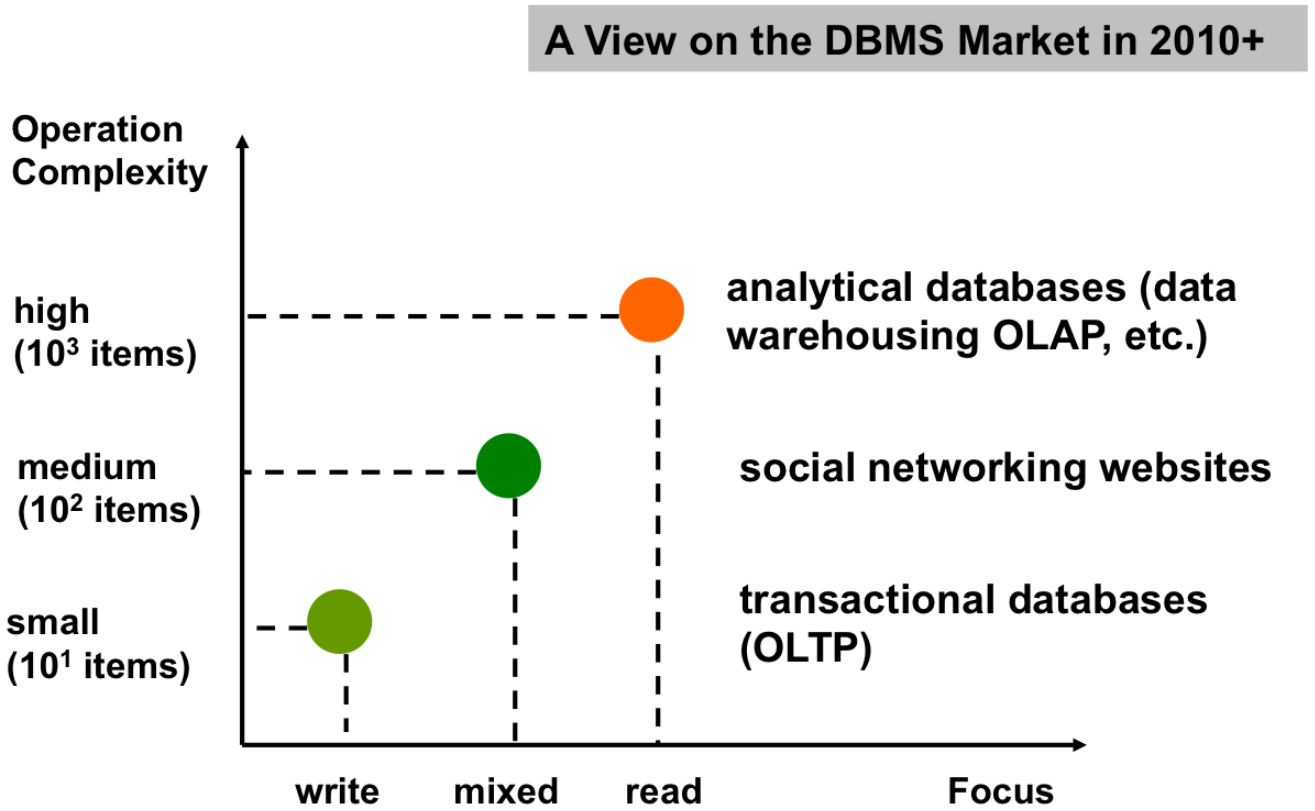
Big Data (6)

In Scientific Contexts

- In science, big data is mostly machine-generated (from extremely large and complex intruments, like the ESO Very Large Telescope, or the CERN Large Hadron Collider).
- The LHC will generate 15 petabytes/year, i.e., 1 million gigabytes per year, i.e., more than 1.7 million dual-layer DVDs, per year.
- Growth rates are even more explosive than in business (until an internet of things becomes reality).
- It is giving birth to a new area, called **XLDB** (for eXtremely Large DataBases).
- Some kinds of social data (e.g., crowdsourcing, citizen science) are also being used but volumes are smaller.

The DBMS Market in 2010+ (1)

The Stonebraker-Cattell Focus v. Complexity Dimensions



The DBMS Market in 2010+ (2)

Traversing the Spaces

- We now have the Vogels landscape and the Stonebraker-Cattell space [Stonebraker and Cattell, 2011].
- We will look at scalable SQL-centric OLTP ideas.
- We will then look at cluster-based approaches to massive parallelism in organizing and analyzing data through SQL-based queries.
- We will look next at the NoSQL approach to massive distribution that underpins the data management strategies of many web giants and some cloud applications.
- Later, we will move to consider a missing Vogels landmark in the Stonebraker-Cattell space, viz., approaches to querying stream data.

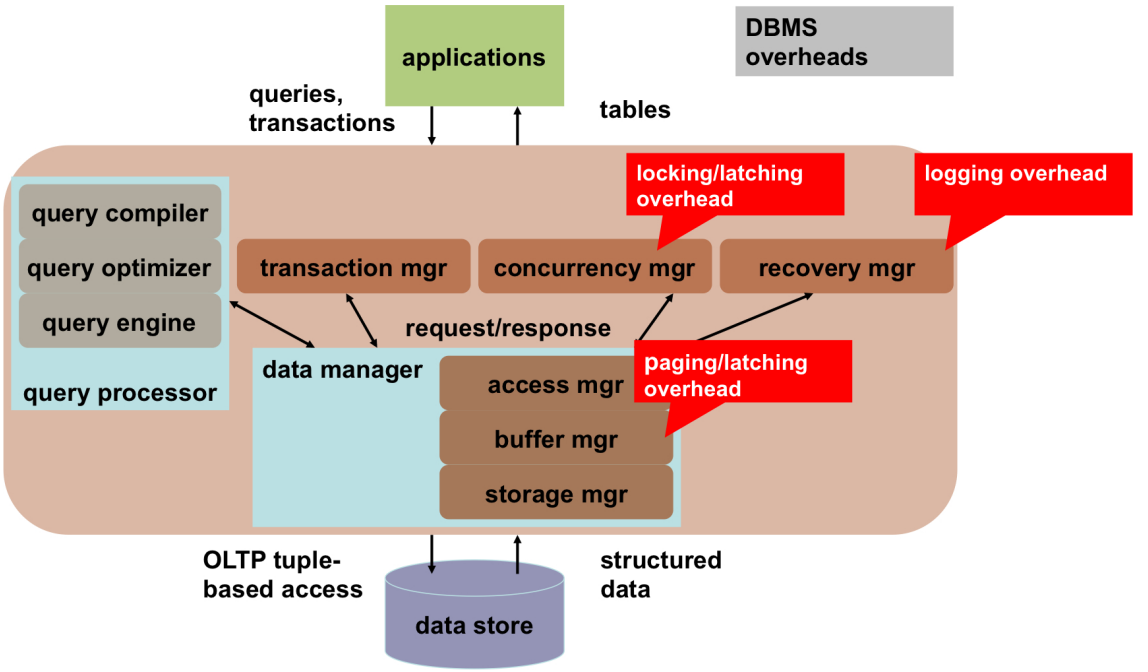
Section 3

Scalable OLTP

Internal Architecture of Classical DBMSs

Four+One Main Service Types

- Query Processing
- Transaction Processing
- Concurrency Control
- Recovery
- Storage Management



Scalable OLTP

From One to Many Through Sharding

- When there is explosive, typically unforeseen, data growth, a single-node DBMS may not be able to cope.
- One cheap-and-dirty fix is to **shard**, i.e., partition the data across many copies of the original single-node DBMS.
- However, many drawbacks ensue from this approach:
  - query operations that touch data across many nodes need to have their semantics enforced inside the application logic, which destroys the declarative-to-procedural contract (and with it, the guarantee of quasi-optimal response times)
  - ditto for update consistency for constraint propagation and maintenance
  - for large numbers of nodes, failure is common, but the logic to detect and fix dangling inconsistencies or out-of-sync replicas is complex and must be done in application logic
  - there is no support for hot swap, so availability is compromised.

Scalable OLTP

The Stonebraker-Cattell Ten Rules

- R1 Look for shared-nothing scalability.
- R2 High-level languages are good and need not hurt performance.
- R3 Plan to carefully leverage main memory databases.
- R4 High availability and automatic recovery are essential for simple-operation scalability.
- R5 Online everything.
- R6 Avoid multi-node operations.
- R7 Don't try to build ACID consistency yourself.
- R8 Look for administrative simplicity.
- R9 Pay attention to node performance.
- R10 Open source gives you more control over your future.

The Stonebraker-Cattell Ten Rules in Detail (1)

R1 Look for shared-nothing scalability

- Shared-memory designs (i.e., a multicore, single-node DBMS over shared primary and secondary memory) suffers from contention and starves the cores forcing designers into sharding.
- Shared-disk designs (i.e., a multicore, single-node DBMS with private primary memory per CPU but sharing secondary memory) suffers from complex buffer and lock management needs which limit scalability.
- Shared-nothing designs (i.e., each node has its own private and secondary memory) are scalable if partitioning is load-balancing and if the DBMS pounces on opportunity for touching as few partitions as possible (ideally one).
- While shared-memory and shared-disk designs do not scale beyond tens of nodes, shared-nothing designs scale to hundreds and are limited only by network bandwidth.

The Stonebraker-Cattell Ten Rules in Detail (2)

R2 High-level languages are good and need not hurt performance

- SQL transactions can be prone to overheads comparing to low-level programming, including:
  - O1: sub-optimal plan selection
  - O2: need for data to flow through DBMS
  - O3: sub-optimal code due to generation
  - O4: generic services for storage management, concurrency and recovery
- Regarding O1, plan selection is more likely to yield a better plan than all but the most skilled and knowledgeable (and costly) programmers.
- Regarding O2, if stored procedures (rather than database connectivity APIs) are used, there's only one over-and-back message.
- Regarding O3, queries are compiled to either native or interpretable code, so the tool chain is essentially the same for handwritten and optimizer-selected query code.
- O4 is addressed in R3 below.

The Stonebraker-Cattell Ten Rules in Detail (3)

R3 Plan to carefully leverage main memory databases

- A shared-nothing design with 16 nodes, each with 64 GB primary memory, is now commonplace.
- This is 1 TB of (non-shared) primary memory, making it possible to deploy many (even most) transactional DBs on primary memory **only**.
- But the profiling of OLTP processing in classical DBMSs shows that only 13% is useful work, the overheads for multithreading, locking, logging, and buffer management consume 11%, 20%, 23% and 33%, resp. [Harizopoulos et al., 2008]
- The magnitude of each overhead is such that removing one or two still allows much performance to leak.
- Removing all of them requires reenginering the classical DBMS architecture for the case of main-memory deployments.

The Stonebraker-Cattell Ten Rules in Detail (4)

R4 High availability and automatic recovery are essential for simple-operation scalability

- DBMSs are expected to never fail to be available.
- However, when there is a massive number of nodes, quasi-random continuous single-node failures are likely.
- Failures caused by software faults are often hard to detect, particularly in massively distributed scenarios.
- Another challenge is the force of the CAP theorem [Brewer, 2000, Gilbert and Lynch, 2002] which states that a distributed system can only exhibit any two (but not all) of three desiderata: **consistency**, **availability**, and **partition tolerance**.
- In other words, that:
  - all nodes see the same data at the same time
  - every request receives a response as to whether it succeeded or failed
  - the system continues to operate despite arbitrary message loss

The Stonebraker-Cattell Ten Rules in Detail (5)

R5 Online everything

- DBMSs are expected to have no down-time.
- This should be so even if one needs to change schemas or indices, add or remove nodes, or upgrade software components.

The Stonebraker-Cattell Ten Rules in Detail (6)

R6 Avoid multi-node operations

- When scaling out, two goals must be met as close as possible:
  - computational load must be split evenly between nodes, which can be fostered by replication, in the case of read-only scalability, and by partitioning, in the case of read/write scalability
  - operations that span more than one node or partition must be rare.
- If load is uneven, contention or lagging will cause performance leakage.
- If operations often span more than one node, cross-node communication and synchronization delays will cause performance leakage.
- Here too the CAP theorem places limit on ambitions for P.

The Stonebraker-Cattell Ten Rules in Detail (7)

R7 Don't try to build ACID consistency yourself (1)

SQL-based DBMSs support ACID guarantees:

Atomicity All the operations in the transaction complete, or none does.

Consistency All the integrity constraints are true when the transaction begins and when it ends.

Isolation No transaction interferes with any other transaction.

Durability If a transaction commits, its effects persist and cannot be reversed by failures or faults.

Non-SQL systems often offer BASE guarantees:

Basically Available by relaxation of isolation,

Soft-state by relaxation of durability and atomicity,

Eventually consistent by optimistic postponement of consistency.

The Stonebraker-Cattell Ten Rules in Detail (8)

R7 Don't try to build ACID consistency yourself (2)

- BASE systems version/timestamp every write, leading to multiplicity in the presence of parallelism, and request the application to decide on how to reconcile.
- They sometimes refrain from updating if the value they meant to change has already changed.
- They can often ACID guarantees on a single object or partition.
- They can decide on postponed consistency resolution on the basis of quorum reads/writes (i.e., before returning, ensure a majority of nodes has completed the read or write).
- In all cases, application programmers have a complex task if updates must be coordinated (e.g., funds transfer).

The Stonebraker-Cattell Ten Rules in Detail (9)

R8 Look for administrative simplicity

- Classical DBMSs have poor out-of-the-box behaviour, leading to dependence on highly-costly professionals to install, tune and run a multi-node system.
- Data appliances (which we consider in a supplement to these notes) are one response to this challenge.

The Stonebraker-Cattell Ten Rules in Detail (10)

R9 Pay attention to node performance

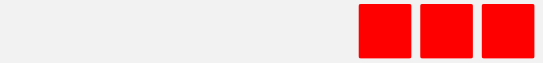
- While its true that, if inherently scalable, one can respond with more nodes to a growth in the problem, if one can improve the individual contribution of each node, then one will need fewer nodes to scale linearly.
- Often performance per node varies by one order of magnitude.
- If a node can perform ten times better, then an installation that needed 500 nodes can be trimmed to 50.
- The savings in acquisition costs, rack space, and energy consumption could be very significant.
- Churn due to individual node failures is likely less as well.

The Stonebraker-Cattell Ten Rules in Detail (11)

R10 Open source gives you more control over your future

- This is an innovation space with start-ups that are born to be sold.
- Being locked into a single supplier can lead to extortionate upgrades and poor-value technical support contracts.
- Open source reduces such risks and builds upon knowledge-contribution from large technical communities.

Summary



21st Century DBMS: Scalability

- The first decades of the 21st century seem likely to be characterized by ever expanding availability of data, touching of tens or hundreds of petabytes per year per domain.
- Data is not just more voluminous, it is also continuously available at ever faster rates, more diverse in structure and more variable in terms of patterns or models that characterize them.
- This makes scalability and flexibility imperative.
- It also becomes crucial to cust the management and operation costs of data management solutions.
- It remains possible to scale out SQL-based DBMSs but great awareness needs to be paid to issues such as codified in the 10 Stonebrake-Cattell rules.

Why Anything Other than SQL?

Different Rates of Growth, Underlying Inflexibility

- There is a perception that the growth in data volumes causes SQL engines to fail to cope with the velocity aspect of big data.
- There is a perception that the variety of big data cannot be accommodated by the relational model underlying SQL engines.
- NoSQL (or NOSQL) engines aim to respond to these perceived shortcomings.
- When people refer to cloud databases, there is often an implicit reference to NoSQL databases, but we will not pursue the cloud aspects of the matter in this course unit.
- Stonebraker and Cattell (among others) argue that these shortcomings may be addressed without renouncing SQL engines and their underlying foundations.

NoSQL or NOSQL? (1)

No? Not Only?

- There is growing consensus on that the slogan is not '**No SQL!**' but rather '**Not only SQL!**'
- Which interpretation is adopted matters because 'NoSQL' postulates that SQL has no role to play, whereas 'NOSQL' postulates that SQL has a role to play but is not, alone, enough.

NoSQL or NOSQL? (2)

Is It Really About SQL?

- But, is SQL (and, by implication, the relational model, calculi and algebra) the right bone of contention or is SQL merely a stand-in for relational DBMS architectures (and OLTP more specifically)?
- Indeed, there seem to be two sources of dissatisfaction:
  - the strictures of the relational model (essentially, the integrity constraints, and not just the domain, key, and referential ones, but the normal forms too, since they coerce the data into regular records),
  - the strictures of how SQL engines coerce applications into running under certain architectural constraints that are perceived as performance leaks (most notably, enforcing ACID transactional semantics) in big data contexts.
- The perception is, on the one hand, that classical OLTP DBMSs cannot linearly scale response time on big data, and, on the other, that big data cannot be coerced into a strict relational view of the world.

NoSQL: What (1)

Six Characteristic Abilities

To [Cattell, 2010] six abilities, together, distinguish NoSQL databases:

1. To horizontally scale the throughput of simple-operation workloads over many (potentially thousands of) servers.
2. To replicate and distribute data (through partitioning) over those (thousands of) servers.
3. To expose a simple call-level interface or protocol (in contrast to a JDBC-like SQL binding).
4. To offer BASE guarantees only (i.e., a weaker concurrency and recovery model than the ACID guarantees offered by SQL-based DBMSs).
5. To make use (non-stop, 24 x 7 x 4 x 12) of distributed indexes efficiently for replication-rich, elastic provision of data storage in both primary and secondary memory.
6. To cope with (non-stop, 24 x 7 x 4 x 12) variations in the structure of objects stored.

NoSQL: What (2)

Breakthroughs

As proofs-of-concept, the following systems have been influential:

1. `memcached.org` showed that indexes held in primary memory could lead to scalability by distribution and replication of objects over many nodes.
2. **Dynamo** [DeCandia et al., 2007] showed that basic availability was possible in the presence of massively-distributed, elastically scalable, highly-replicated storage if applications did not require stricter guarantees than eventual consistency.
3. **Bigtable** [Chang et al., 2008] showed that persistent extensible record stores could scale to thousands of nodes.

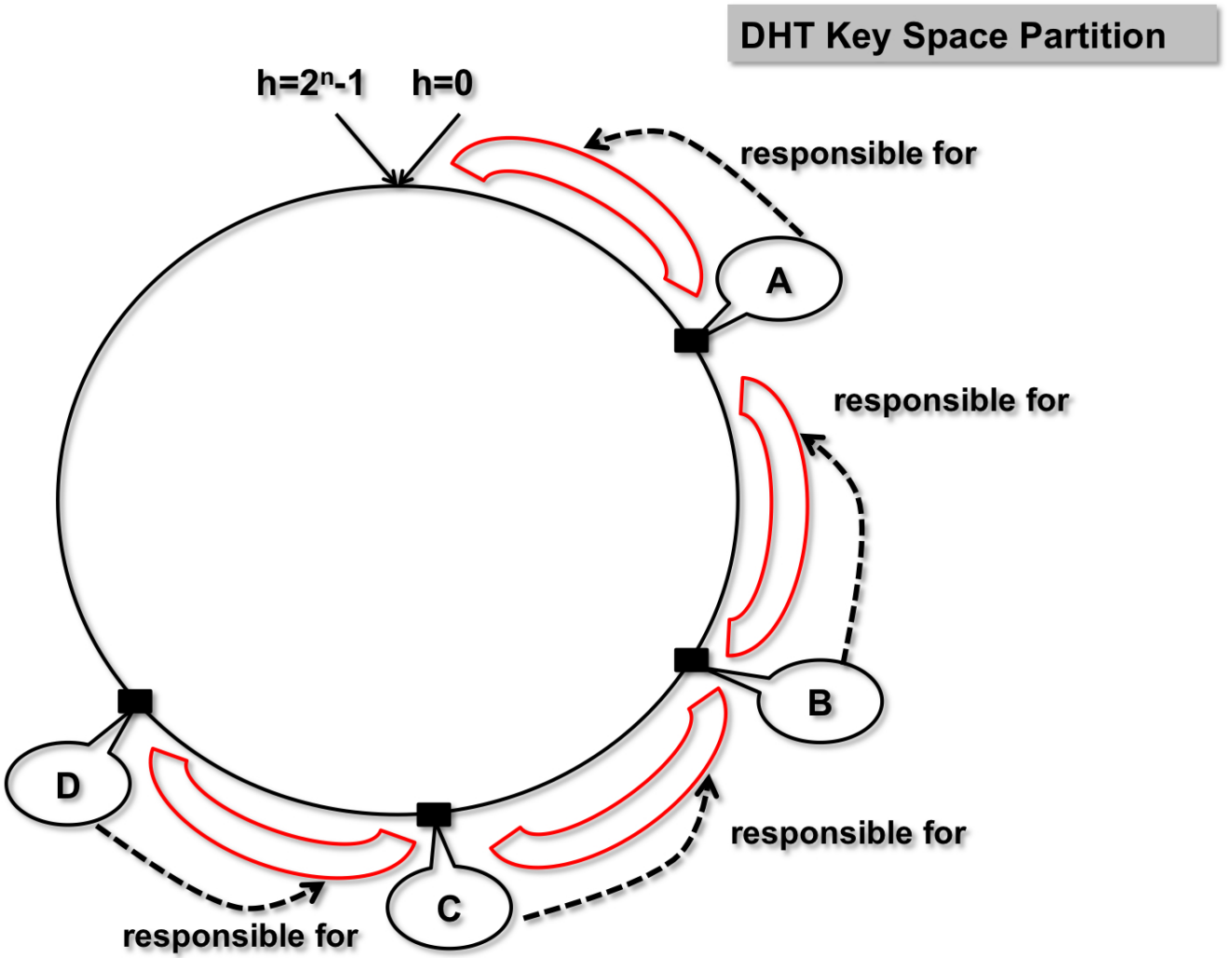
Ring-based DHTs for Scalability (1)

DHTs as a Distributed Storage Structure

- (Ring-based) distributed hash tables (DHTs) implement distributed storage schemes by partitioning the stored objects.
- Each key-value pair (k, v) is stored in the server at position $h(k)$, where the range of h is $[0, 2^n - 1]$ where n is the number of servers.
- This is reminiscent of hash-based partitioning we have seen before.
- The API, as we have seen before, is `put(key, value)`, `get(key):value`.

Ring-based DHTs for Scalability (2)

Key Space Partitioning in Ring-Based DHTs



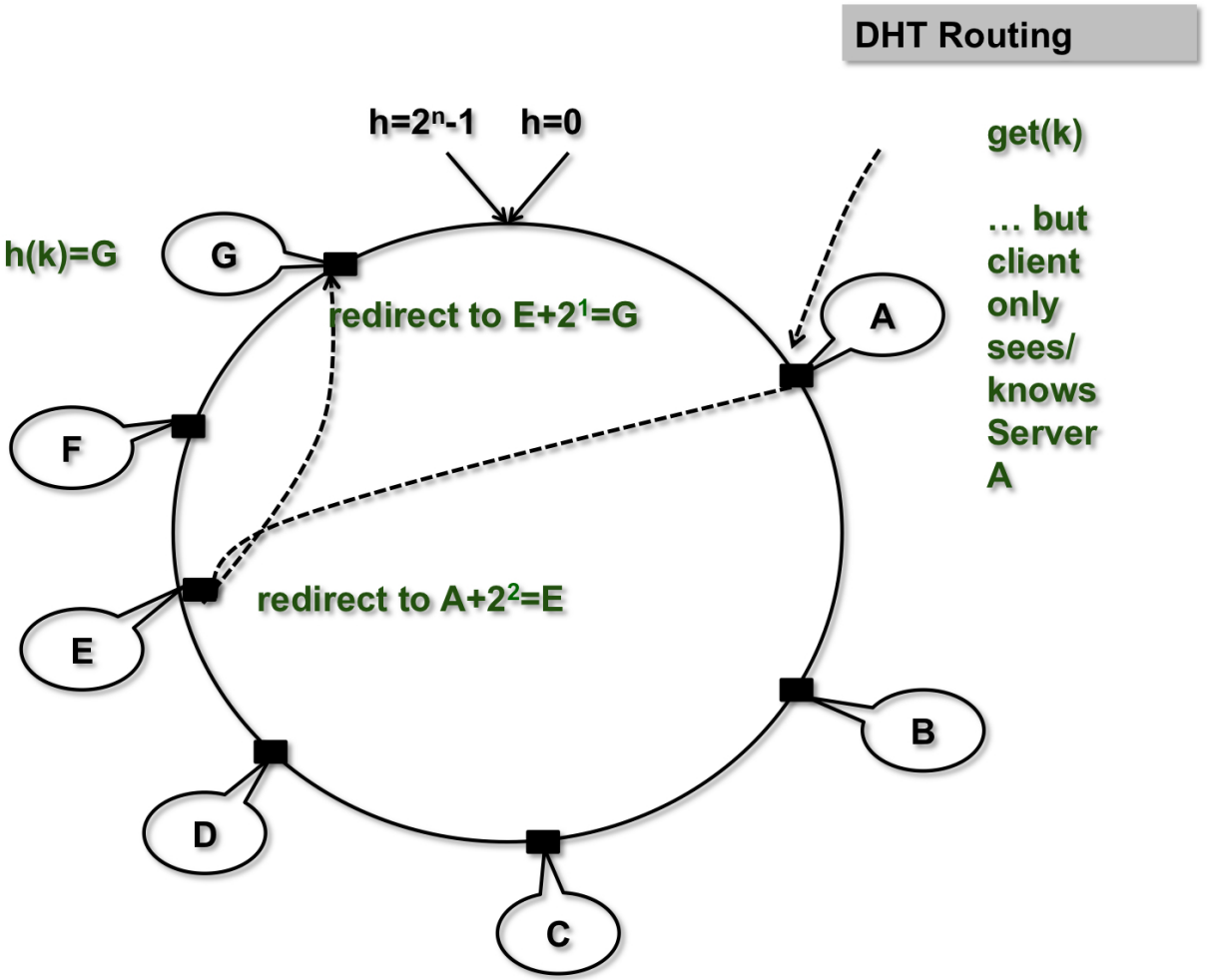
Ring-based DHTs for Scalability (3)

Problems to Be Solved

1. How do we route an operation like get(k) to the right server, i.e., $h(\mathtt{k})$? A client only knows one server. It does not (and should not) necessarily know who is $h(\mathtt{k})$ (i.e., what its position is).
2. How do we avoid disruption when n grows or shrinks to $n \pm 1$? We should not need to reorganize the whole storage space (i.e., cause data movement everywhere).
3. How do we replicate for availability? We want the solutions to the two previous problems to also work when data is replicated.

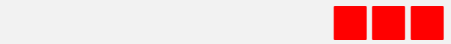
Ring-based DHTs for Scalability (4)

Routing (1)



Ring-based DHTs for Scalability (5)

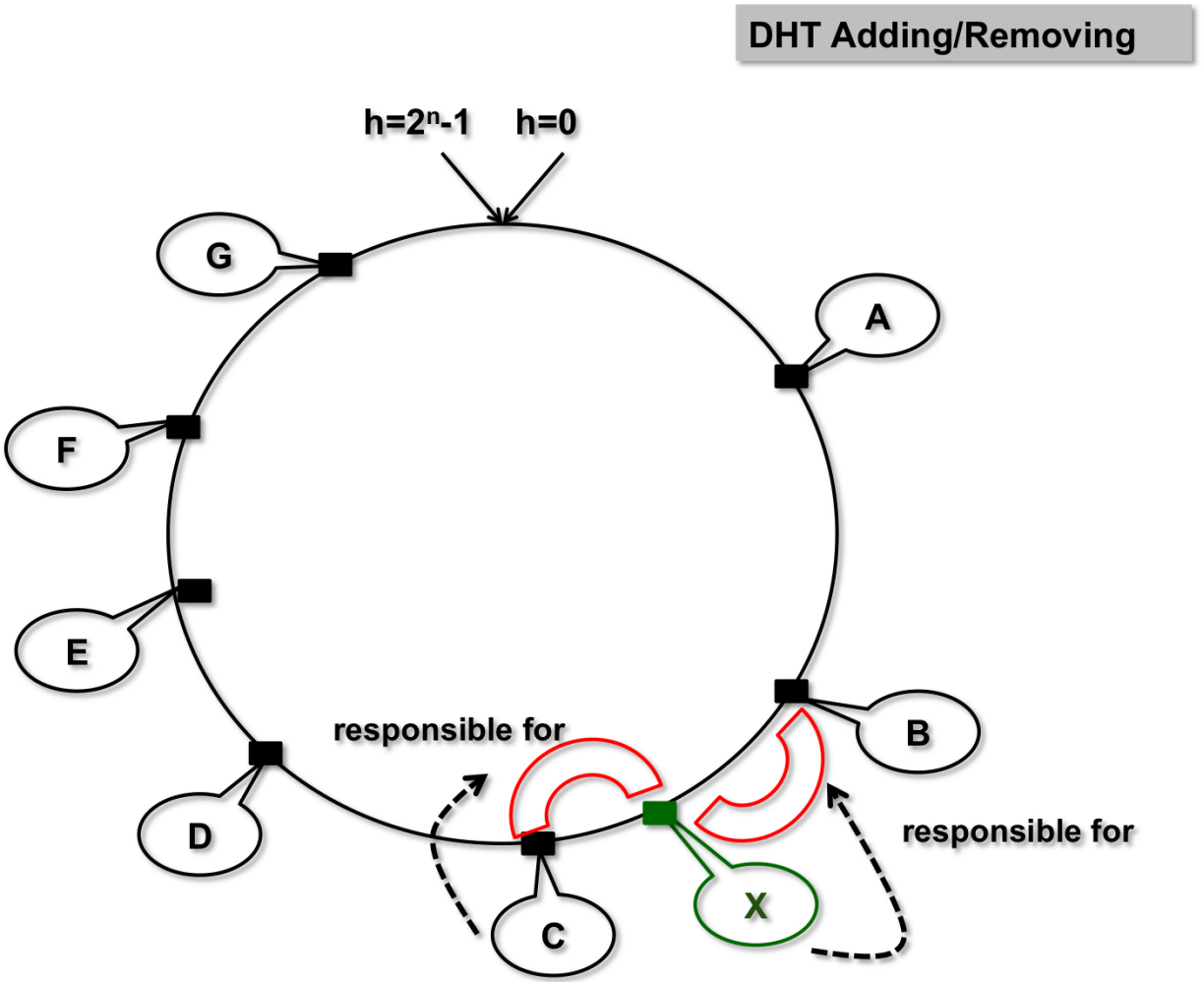
Routing (2)



- The naïve way to route is for servers to know/store who their one-hop neighbours are.
- When you get a request you cannot serve, route it to your (clockwise) next-hop neighbour.
- This is $O(n)$, so we must try to do better.
- To do better, we memorize a so-called **finger table**, whereby a server a knows/stores the positions of $a, a + 2^0, a + 2^1, \ldots, a + 2^{n-1}$.
- When you get a request you cannot serve, route it to the node in your finger table that is (clockwise) closest to the destination $h(k)$.
- This is $O(log(n))$, so we will do better.

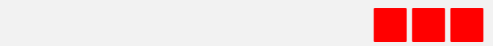
Ring-based DHTs for Scalability (6)

Adding/Removing Servers (1)



Ring-Based DHTs for Scalability (7)

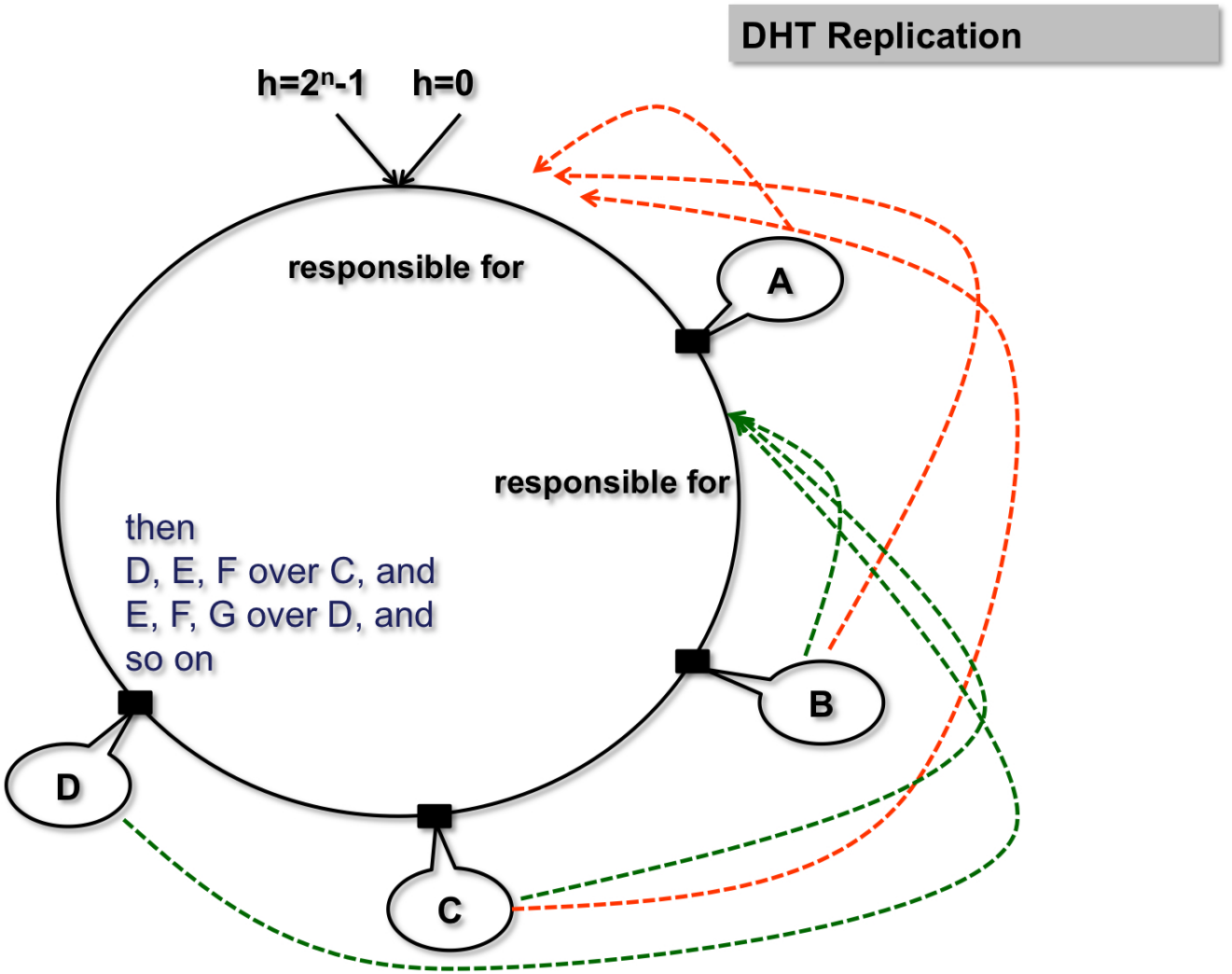
Adding/Removing Servers (2)



- To add a new Server X:
  - Assign a random (unassigned) value in the key space as the position of X.
  - Say X then lies between Server B and Server C.
  - Call B the (clockwise) last-hop neighbour of X, and C the next-hop.
  - Keys between the new server (X) and its next-hop neighbour (C), remain with C.
  - Keys between the new server (X) and its last-hop neighbour (B), become the responsibility of the X, the new server.
- To remove a server, its keys become the responsibility of its next-hop neighbour.
- Note that all changes are limited to a one-hop neighbourhood, i.e., there is minimal disruption and overhead.

Ring-based DHTs for Scalability (8)

Replication in Ring-Based DHTs (1)



Ring-based DHTs for Scalability (9)

Replication in Ring-Based DHTs (2)

- Replication is desirable since it increases the resilience to failure, avoids contention due to bottlenecks, and increases opportunities for parallelism.
- Ring-based DHTs allow for elegant replication (in the sense that the solutions to routing and reconfiguration need only obvious extensions in the presence of replication).
- If r is the degree of replications, then we assign a key k to the servers at position $h(k)$, $h(k)+1$, $\ldots$, $h(k)+r-1$.

Approaches to Transactional Guarantees (1)

ACID Effects on Availability and Scalability

- Guaranteeing ACID can be expensive due to the need to pessimistically log, lock, and latch.
- Most DBMSs use the classical two-phase commit protocol (2PC), which is succinctly illustrated in [Alberton, 2011], to enforce ACID properties on transactions.
- 2PC relies on a central coordinator and so has a single-point of failure (compromising availability) and a performance bottleneck (compromising scalability).
- A distributed alternative exists, viz., the Paxos algorithm.
- Megastore [Baker et al., 2011] uses it to implement a NoSQL store with ACID guarantees.

Approaches to Transactional Guarantees (2)

The BASE Alternative

- The BASE alternative to ACID optimistically avoids locking and latching and pays the price of not always being in a consistent state.
- The need for locking and latching is because of hard state (i.e., destructive assignment).
- If we adopt a notion of soft state (e.g., allow the coexistence of different versions of the state of a variable, or attribute), then we can postpone the resolution of conflicts.
- Unfortunately, this also means that only the application can resolve conflicts (on which, recall Stonebraker-Cattell Rule 7).

Vector Clocks for Soft-State Eventual Consistency (1)

First, Multiversion Concurrency Control (1)

- Multiversion concurrency control (MVCC) introduced the idea of storing timestamped information about read and write events in order to avoid locking pessimistically.
- Thus, rather than destructively assigning, one sees a history of events, of which some are accesses and some are changes.
- Each transaction gets a timestamp.
- There are multiple transactions.
- A write by a transaction with timestamp t cannot complete if there is any running transaction with a timestamp earlier than t.

Vector Clocks for Soft-State Eventual Consistency (1)

First, Multiversion Concurrency Control (2)

- Objects have read timestamps.
- If a transaction T with timestamp t' tries to write onto an object o with a read timestamp $t > t'$, then T is aborted and restarted with a new timestamp.
- Otherwise, i.e., if $t < t'$, then T is allowed to write, o is versioned into o' and the read and write timestamps of o' are set to t'.
- MVCC incurs the cost of storing multiple versions of objects.
- MVCC enables application to reap the benefits of non-blocking reads.

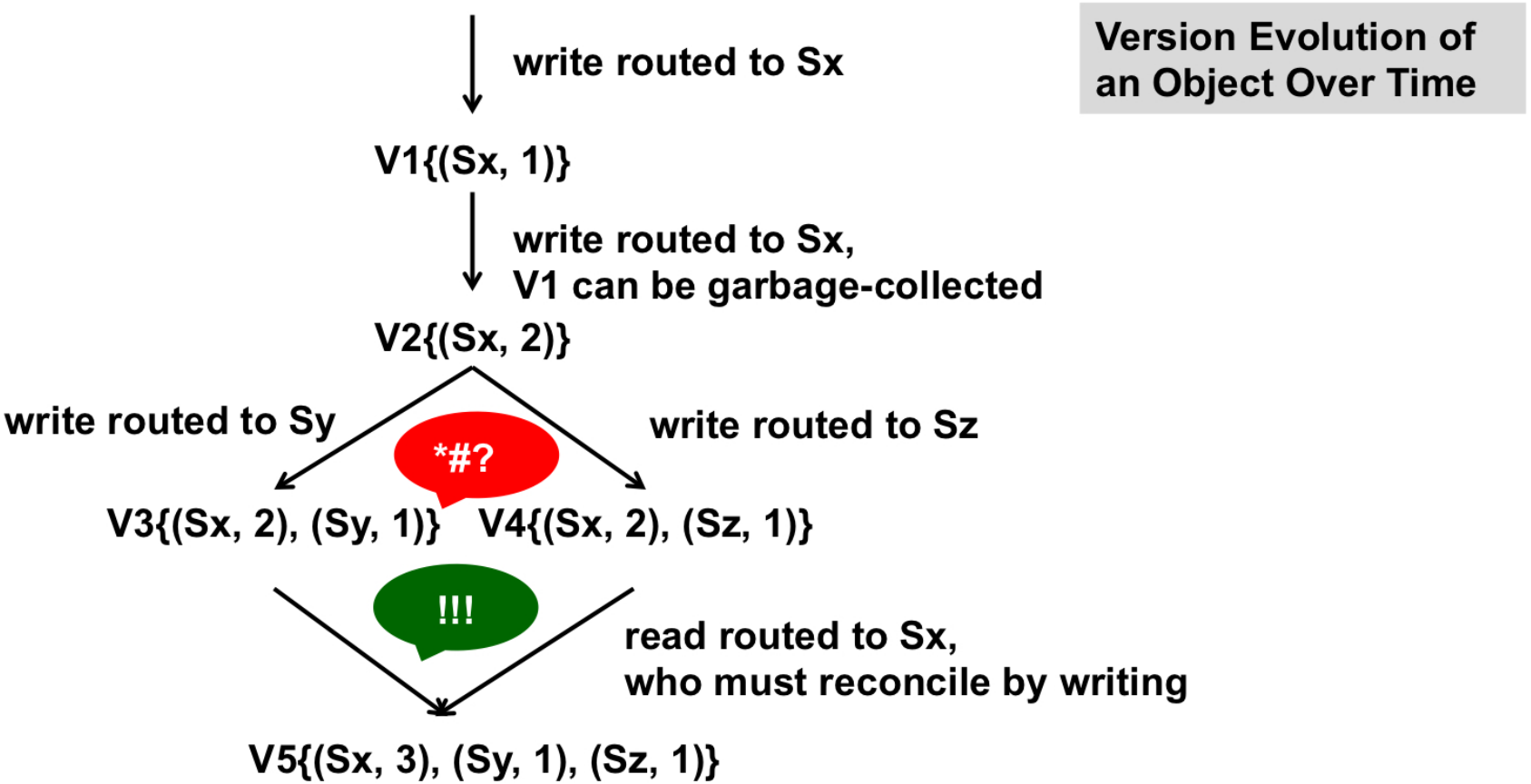
Vector Clocks for Soft-State Eventual Consistency (2)

A Variation on MVCC Timestamps

- Dynamo [DeCandia et al., 2007] uses vector clocks to establish whether different versions of the same object are causally related.
- A vector clock is represented as a list of (node, counter) pairs, rather than just a timestamp as in MVCC.
- One vector clock is associated with every version of every object.
- We can tell that two versions of an object are on parallel branches or else are causally related by examining their vector clocks.
- If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten (e.g., garbage-collected.)
- Otherwise, the two changes are considered to be in conflict and require reconciliation.

Vector Clocks for Soft-State Eventual Consistency (3)

An Example (1)



Vector Clocks for Soft-State Eventual Consistency (4)

An Example (2)

- A client writes a new object.
- The node (say Sx) that handles the write for this key increases its counter and uses it to create the data's vector clock.
- The system now has the object D1 and its associated clock [(Sx, 1)].
- The client updates the object.
- Assume the same node handles this request as well.
- The system now also has object D2 and its associated clock [(Sx, 2)]. D2 descends from D1 and therefore over-writes D1, however there may be replicas of D1 lingering at nodes that have not yet seen D2.

Vector Clocks for Soft-State Eventual Consistency (5)

An Example (3)

- Let us assume that the same client updates the object again and a different server (say Sy) handles the request.
- The system now has data D3 and its associated clock [(Sx, 2), (Sy, 1)].
- Next assume a different client reads D2 and then tries to update it, and another node (say Sz) does the write.
- The system now has D4 (a descendant of D2) whose version clock is [(Sx, 2), (Sz, 1)].

Vector Clocks for Soft-State Eventual Consistency (6)

An Example (4)

- A node that is aware of D1 or D2 could determine, upon receiving D4 and its clock, that D1 and D2 are overwritten by the new data and can be garbage collected.
- A node that is aware of D3 and receives D4 will find that there is no causal relation between them.
- In other words, there are changes in D3 and D4 that are not reflected in each other.
- Both versions of the data must be kept and presented to a client (upon a read) for semantic reconciliation.

Vector Clocks for Soft-State Eventual Consistency (7)

An Example (5)

- Now assume some client reads both D3 and D4 (the context will reflect that both values were found by the read).
- The read's context is a summary of the clocks of D3 and D4, namely [(Sx, 2), (Sy, 1), (Sz, 1)].
- If the client performs the reconciliation and node Sx coordinates the write, Sx will update its sequence number in the clock.
- The new data D5 will have the following clock: [(Sx, 3), (Sy, 1), (Sz, 1)].

When to Use NoSQL? When to Use SQL? When to Use NOSQL

Shifting, Ever-Changing Technological Landscape

- The first thing to note is that, for now, it is false that **one size fits all**, i.e., no single approach is valid for all users/applications.
- SQL and NoSQL are complementary:
  - SQL can scale on big data and retain ACID guarantees for OLTP.
  - Queries are crucial in OLAP productivity too.
  - NoSQL is more flexible to cope with data variety and may pay less overheads due to BASE guarantees only.
  - However, NoSQL dumps the complex semantics that SQL takes on itself on application programmers.
- We still need some way of dealing with machine-generated and physical-world streamed data.

Summary

21st Century DBMSs: NoSQL (or NOSQL)

- In spite of terminological confusion, distributed storage systems that scale horizontally over flexible data types and deliver excellent availability at the cost of soft-state eventual (and application-delegated) consistency are now significant players in the data management landscape.
- NoSQL stores seem complementary to SQL stores, so NOSQL is probably the most appropriate reading.
- The loss of queries is significant, so much so that there are initiatives in place to endow NOSQL stores with a QL of their own.

Acknowledgements (1)

The material presented mixes original material by the author as well as material adapted from

- [Stonebraker and Cattell, 2011]
- [Cattell, 2010]
- [DeCandia et al., 2007]

The author gratefully acknowledges the work of the authors cited while assuming complete responsibility any for mistake introduced in the adaptation of the material.

Lecture 23

Massively-Parallel Schemes: The Map-Reduce Model

Section 1

Cluster-Based Data Management

Cluster-Based Data Management

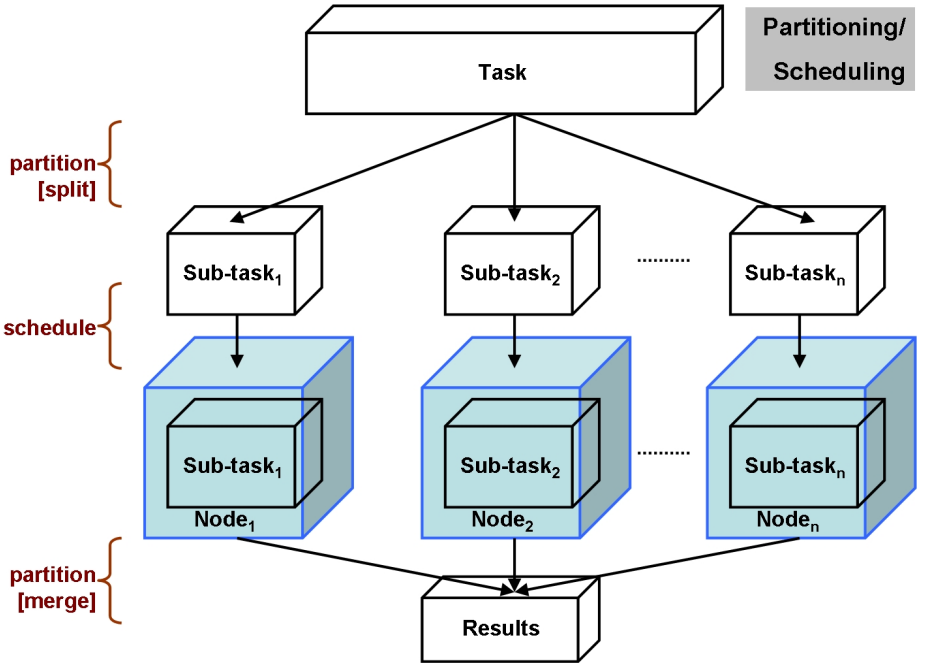
Advanced for Keyword-Based Querying, Incipient for Anything More Complex

- For partitioning and scheduling to be simple enough to carry out in a generic fashion, the parallel and distributed tasks must be even more stringently constrained than relational operators.
- The tasks involved in keyword-based search are an example.
- The tasks involved in frame-based image processing are another.
- In this respect, one of the major difficulties in query processing is data dependency (e.g., partitioning is sometimes attribute- and location-sensitive).

Cluster-Based Computing (1)

Partitioning and Scheduling (1)

- The grain of parallelization depends on the grain at which we can split a task into multiple independent units.
- The scheduling problem then is to assign sub-tasks to processing elements (i.e., threads, or nodes).
- Once sub-tasks are finished, the partitions are merged back into complete results.



Cluster-Based Computing (2)

Partitioning and Scheduling (2)

- The decision space is complex, and requires answers to the following questions (among others):
  - What if we cannot split the tasks into independent sub-tasks?
  - How do we assign sub-tasks to nodes?
  - What if we have more sub-tasks than nodes?
  - How do we merge sub-results into results?
  - How do we know all the nodes have finished?
- The common difficulty in all these is the need for nodes to communicate and access shared resources.
- It is crucial to achieve task independence by avoiding sharing state.

Cluster-Based Computing (3)

Partitioning and Scheduling (3)

Example (Expression-Level Independence)

- When independent branches of an algebraic expression commute and do not access (i.e., make no reference to) shared state, they can be executed in any order.
- In the following expression

  ```
  x := (a * b) + (y * z)
  ```

  since + commutes, a compiler (or a CPU scheduler) can execute the left- and the right-hand side in any order.

Example (Function-Level Independence)

- This is also true, but harder to automate, at the level of entire functions.
- In the following expression

  ```
  x := foo(a) + bar(b)
  ```

  if both `foo(_)` and `bar(_)` do not access shared state, they can be parallelized, with the assignment to `x` becoming the merge/synchronization point that blocks waiting for them to complete.

Cluster-Based Computing (4)

Detecting Dependencies Automatically is Hard

- In the last example, there is independence between `foo(_)` and `bar(_)` and dependence between each one of them and `x`.
- For a partitioner/scheduler infrastructure, detecting these properties is hard.
- The inability to do so prevents massive parallelization at the coarse grains that are needed to process massive volumes of data.

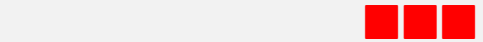
Cluster-Based Computing (5)

When It Can Be Done

- When task-level independence is ensured, in cluster-based computing, a single-master, many-slaves approach can be employed:
  1. The master initially owns the data.
  2. The master spawns several workers to process the partitions it creates.
  3. The master waits for the workers to complete before merging results.
- Daisy-chaining leads to workers that consume from and produce for other workers.
- Queues can be used to allow $n : m$ relationships between producing and consuming workers.
- In a distributed cluster, location transparency for both tasks and data is necessary.

Cluster-Based Computing (6)

But How Can It Be Done?



- One approach to circumventing this obstacle is to limit the expressiveness of the computation performed by individual tasks in such a way as to ensure task independence.
- One computational model that has acquired prominence recently (due to its widespread use in Google, Yahoo and other web-based companies) is referred to as **the map-reduce computational model**.

Section 2

The Map-Reduce Computational Model

The Map-Reduce Computational Model (1)

Roots in Functional Programming (1)

- The map-reduce model takes its name from its roots in the functional programming paradigm (of which concrete examples are Haskell, Standard ML and Miranda, among others).
- In (pure) functional programs, functions are side-effect free.
- Recall our previous examples: in functional languages, two functions used in the same expression are guaranteed not to share any state.
- Instead of side-effecting shared data structures, functions in pure functional programs create completely new ones.
- They are, therefore, inherently parallelizable.

The Map-Reduce Computational Model (2)

Roots in Functional Programming (2)

- Consider a function `foo` that takes a list of numbers, computes a sum of its elements and adds to the latter the length of the input list.
- The following definition (in Haskell syntax) uses the `sum` and `length` primitives:

```
>> let foo x = sum x + length x
>> foo [1,2,3]
9
```

- Now, consider a function `bar` that takes a list of numbers and computes the square of applying `foo` to it.
- The definition (in Haskell syntax) could be:

```
>> let bar x = foo x * foo x
>> bar [1,2,3]
81
```

The Map-Reduce Computational Model (3)

Roots in Functional Programming (3)

- Since neither `foo` (or the primitives it relies on) nor `bar` interfere with another, if they appear as operands of a binary commutative function (such as +), they can be evaluated in any order:

```
>> foo [1,2,3] + bar [1,2,3] == bar [1,2,3] + foo [1,2,3]
True
```

- We can define a function `append` that takes an element and a list and appends the element to the end of the list in the following (convoluted way) without ever relying on shared state:
  1. Reverse the input list
  2. Stick the input element at the front of the results
  3. Reverse the latter result and return
- The definition in Haskell could be:

```
>> let append l ls = reverse (l : rev) where rev = reverse ls
>> append 1 [3,2]
[3,2,1]
```

The Map-Reduce Computational Model (4)

Roots in Functional Programming (4)

- Another important characteristic of functional programming languages is that functions can be passed as parameters to other functions.
- In particular, we can define **second-order functions**, i.e., functions that take functions as arguments.
- For example, we can define a function that takes a function and an argument, and applies the former to the latter twice.

```
>> let twice f x = f (f x)
>> twice (+ 1) 1
3
>> twice (* 3) 3
27
```

- In the examples, `(+ 1)` is a function that adds 1 to the argument it is applied to, and `(* 3)` is a function that triples the argument it is applied to.

The Map-Reduce Computational Model (5)

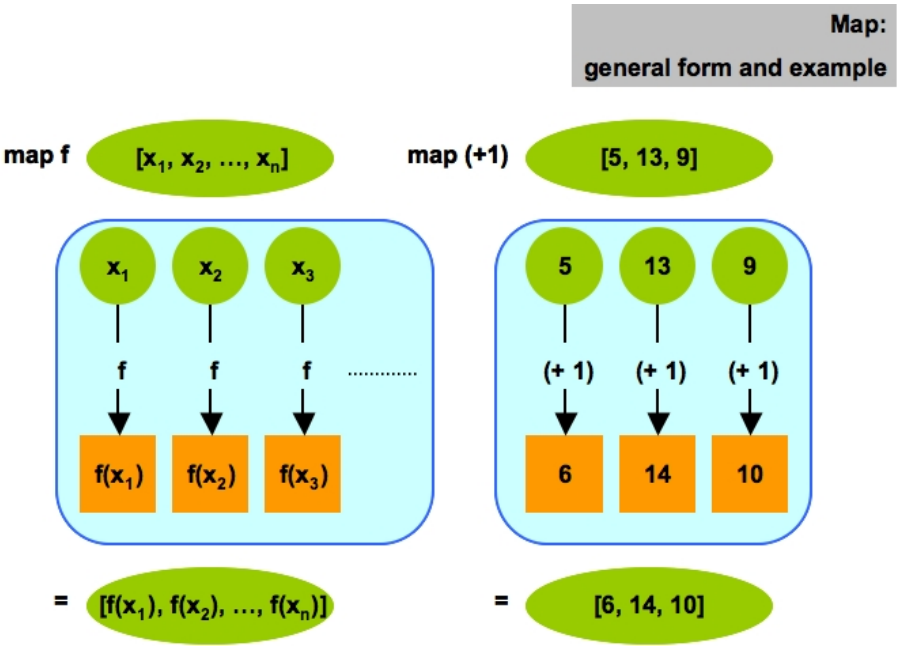
Roots in Functional Programming (5)

- The map-reduce computational model has this name because it is centred around two higher-order functions that are well-known and well-studied in functional programming, viz., **map** and **fold** (sometimes also referred to as **reduce**).
- **map** takes as arguments a unary function and a list, applies the former to each element of the latter, and returns the resulting list.
- **fold** takes as arguments a binary and associative function, a value (interpreted as the initialization of an accumulator) and a list.
- It traverses the list applying the input function to each element and the current value of the accumulator, updating the latter each time.
- It returns the final value in the accumulator once it reaches the end of the list.

The Map-Reduce Computational Model (6)

Map and Fold(s): Examples (1)

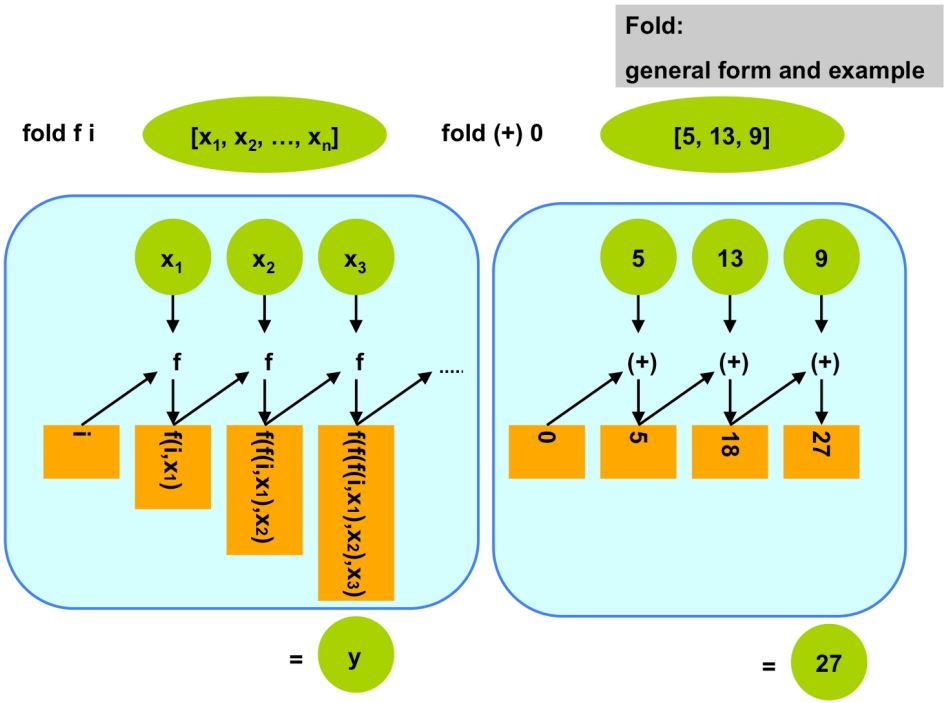
- The general form taken by the application of a mapper function to a list is shown in the figure.
- To the right, a function (+ 1) that increments the elements of a list by one is applied.
- Note that the function could be Boolean-valued (i.e., a predicate), e.g., ($>$ 3).
- Does this suggest $\sigma_{x>3}(R)$ to you?



The Map-Reduce Computational Model (7)

Map and Fold(s): Examples (2)

- The general form taken by the application of a reducer function to a list is shown in the figure.
- To the right, a function (+) that adds the elements of a list (with the initializer set to zero) is applied.
- Does this suggest $\gamma_{SUM}(R)$ to you?
- The fold operation has two versions: `foldl` moves left-to-right in the list, `foldr` moves right-to-left.
- They are equivalent for commutative operations (e.g., addition), but not for non-commutative ones (e.g., subtraction).



The Map-Reduce Computational Model (8)

Map and Fold(s): Examples (3)

```
>> map (+ 2) [3, 4, 3, 8, 4, 1]
[5,6,5,10,6,3]
>> map (> 3) [3,2,4,6,8]
[False,False,True,True,True]
>> foldl (+) 0 [3, 4, 3, 8, 4, 1]
23
>> sum [3, 4, 3, 8, 4, 1] == foldl (+) 0 [3, 4, 3, 8, 4, 1]
True
```

The Map-Reduce Computational Model (9)

Map and Fold(s): Examples (4)

```
>> foldl (+) 0 (map (+ 2) [3, 4, 3, 8, 4, 1])
35
>> foldl (*) 1 (map (+ 2) [3, 4, 3, 8, 4, 1])
27000
```

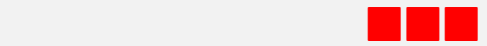
The Map-Reduce Computational Model (10)

Map and Fold(s): Examples (5)

```
>> foldl (*) 1 [3, 4, 3, 8, 4, 1]
1152
>> sum [3, 4, 3, 8, 4, 1] + foldl (*) 1 [3, 4, 3, 8, 4, 1]
1175
>> foldl (*) 1 [3, 4, 3, 8, 4, 1] + sum [3, 4, 3, 8, 4, 1]
1175
>> (sum [3, 4, 3, 8, 4, 1] + foldl (*) 1 [3, 4, 3, 8, 4, 1])
    == (foldl (*) 1 [3, 4, 3, 8, 4, 1] + sum [3, 4, 3, 8, 4, 1])
True
>> (foldl (-) 0 [3, 4, 3, 8, 4, 1]) == ((((((0-3)-4)-3)-8)-4)-1)
True
>> (foldr (-) 0 [3, 4, 3, 8, 4, 1]) == (3-(4-(3-(8-(4-(1-0))))))
True
>>
```

The Map-Reduce Computational Model (11)

Implicit Parallelism in `map`



- In functional programming, applying the mapper function to an element does not have any impact on the result of applying the same mapper function to any other element
- If the order of application of the mapper is immaterial, it can be applied in parallel.
- The map-reduce computational model exploits this to the full.
- This is the way in which it limits the expressiveness of the computation performed by individual tasks so as to ensure task independence and hence be free to execute tasks in massively-parallel mode.

Summary

21st Century DBMSs: Massively-Distributed Data Processing (1)

- Cluster-based schemes hold the promise of complete reliance on commoditization.
- By constraining the computational model, it is possible to run massively-parallel computations without application developers having to design their solutions in parallel terms.
- One such computational model, founded on functional programming, centres around the well-known, well-studied **map** and **fold** higher-order functions.

Lecture 24

Massively-Parallel Schemes: Map-Reduce Engines

Section 1

Motivation and Functionalities

Map-Reduce Engines (1)

Motivation

- The motivation is to process terabytes of data in parallel across thousands of processing elements without requiring the application developer to write any other code than mapper and reducer functions.
- The best-known map-reduce engine is Google's MapReduce/GFS.
- GFS stands for Google File System, which is the distributed storage-level infrastructure relied upon in Google's MapReduce engine.
- Yahoo has contributed an open-source version to the Apache Software Foundation, called Hadoop/DFS.

Map-Reduce Engines (2)

Functionalities

- A map-reduce engine automatically parallelizes and distributes the execution of the mapper and reducer functions that characterize an application.
- Hence, it is an infrastructure for automatic partitioning and scheduling.
- It is highly-fault-tolerant, i.e., it is extremely resilient when nodes (or tasks in nodes) fail.
- It provides monitoring tools and exports a simple (if restricted) abstraction to application developers.

Section 2

Programming/Execution

Map-Reduce Engines (3)

The Programming/Execution Model (1)

- The programming model (i.e., the abstractions with which an application developer designs a solution to a problem) uses the functional programming approach we described above.
- Developers implement two functions with the following interfaces:

```
map ( in_key, in_value                    )
  -> (out_key, intermediate_value) list

reduce (out_key, intermediate_value list)
    -> out_value list
```

Map-Reduce Engines (4)

The Programming/Execution Model (2)

- The mapper function is fed data items (e.g., lines out of a file, rows out of a database) in the form of (in_key, in_value) pairs.
- The role of the mapper function is to transform each one of the latter into an (out_key, intermediate_value) pair.
- Note that the map-reduce engine requires the mapper to:
  1. apply to one element only
  2. associate to the result an output key
- The results of the map phase are held in a **barrier**, which is where intermediate values are left waiting.
- By removing from the writer of the mapper function, any control over the grain of the task, a map-reduce engine can take decisions on task partitioning and parallel scheduling.

Map-Reduce Engines (5)

The Programming/Execution Model (3)

- The engine waits for the map phase to finish.
- When it is, all the intermediate values for a given output key are available in the barrier where they are combined to form a list per output key.
- The reducer function is fed such lists into one or more (typically one) final value for that same output key.
- By causing the writer of the mapper function to assign an output key, a map-reduce engine creates an opportunity to parallelize the reduction phase too, since each reduction task can operate on lists associated with different output keys.

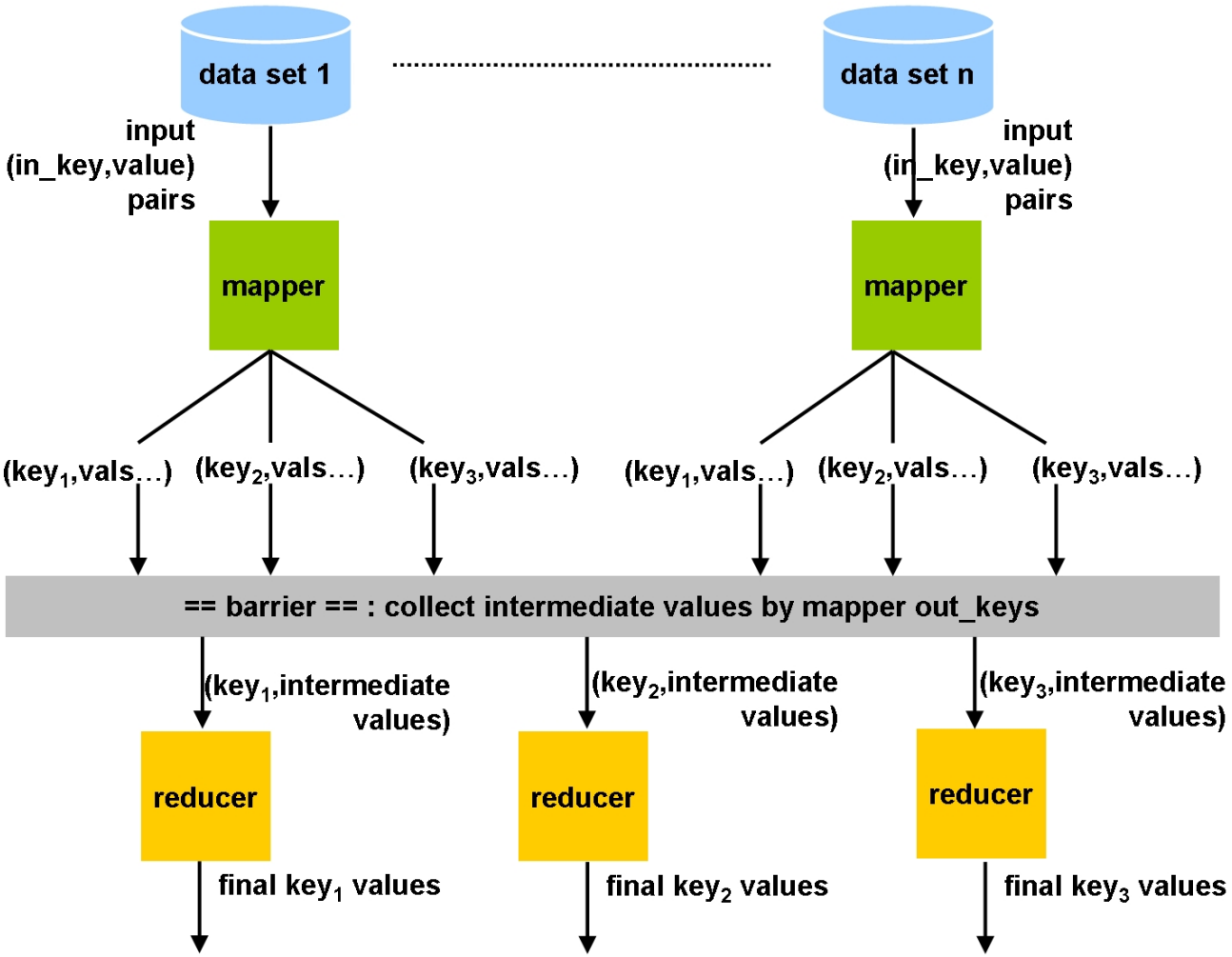
Map-Reduce Engines (6)

The Programming/Execution Model (4)

- As far as implicit parallelization goes:
  - the mapper functions run in parallel, creating different intermediate values from different input data sets and emitting them with an output key, with which the engine can parallelize the reduce functions;
  - the reducer functions run in parallel, operating on different output key values.
- The only bottleneck is that the reduce phase cannot complete until the map phase is completed.

Map-Reduce Engines (7)

An Example Map-Reduce Computation



Map-Reduce Engines (8)

Example Mapper and Reducer from Google: Counting Word Occurrences

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
 Emit(AsString(result));
```

Section 3

Locality, Fault Tolerance, Optimizations

Map-Reduce Engines (9)

Locality Effects

- The master program divvies up tasks based on where data is located.
- For example, map tasks preferably run on the same node where the data file is, or at least in the same rack.
- Map tasks are given 64-MB blocks (which is the size of the chunks served by the underlying Google File System).

Map-Reduce Engines (10)

Fault Tolerance

- The master detects worker failures and
  1. Re-executes completed and in-progress map tasks.
  2. Re-executes in-progress reduce tasks.
- The master detects when an (in_key, in_value) pair causes a map task to crash and skips that when re-executing.

Map-Reduce Engines (11)

Some Optimizations

- Since the reduce task cannot complete until the map task is complete, a single slow component can rate-limit the whole process.
- Therefore, the master replicates slow-going map tasks in several components, keeps the first to finish and ignores the rest.
- So-called combiner tasks can be scheduled in the same machine as a mapper to perform a mini-reduce and save bandwidth before the real reduce phase starts.

Lecture 25

Query Processing with Map-Reduce

Section 1

A Worked-Out Motivating Example

Map-Reduce in Query Processing (12)

A Worked-Out Example (1)

- Assume that a relation with the following schema:

  `Employees (emp_no:int, dept_name:str, salary:real)`

  has been horizontally fragmented per department name and that the latter has domain {'sales', 'manufacturing'}.
- Thus, the fragments are:

$$Employees_1 \leftarrow \sigma_{dept\_name='sales'}(Employees)$$

$$Employees_2 \leftarrow \sigma_{dept\_name='manufacturing'}(Employees)$$

and, therefore, `Employees` can be reconstructed as:

$$Employees \leftarrow Employees_1 \cup Employees_2$$

Map-Reduce in Query Processing (13)

A Worked-Out Example (2)

- Consider now the following SQL query over the distributed database above:

```
SELECT     SUM(A.salary)
FROM       Employees A
WHERE      A.salary > 1000
GROUP BY   A.dept_name
```

Map-Reduce in Query Processing (14)

A Worked-Out Example (3)

```
map(String input_key, TupleSet input_value):
//          input_key: a fragment-defining attribute value
//        input_value: the tuples in that fragment
//         output_key: the GROUP BY attribute value
// intermediate_value: aggregation attribute value
//                     that satisfies the WHERE clause
for each tuple T in input_value:
  if  T.salary > 1000
        EmitIntermediate(T.dept_name, T.salary);

// the barrier contains all (group-name, group-member) pairs
```

Map-Reduce in Query Processing (15)

A Worked-Out Example (4)

```
// reduce gets (group-name, group-member) pairs from the barrier
// as if hashed buckets

reduce(String output_key, Iterator intermediate_values):
  //   output_key: the group-by attribute value
  // output_value: the partition contents defined by it
  int result = 0;
  for each v in intermediate_values:
    result += v;
Emit((output_key, result));
```

Map-Reduce in Query Processing (16)

A Worked-Out Example (5)

- The analysis of the above functions in terms of relational algebra is as follows.
- In this case, the mapper implements the following query:

$$B \leftarrow \pi_{dept\_name,salary}(\sigma_{salary>1000}(Employees_j))$$

- Because `Employees` is fragmented on `dept_name`, the parallelization criterion has already been made explicit, as it were.
- In relational terms, the barrier holds the result of running the above query in parallel over each fragment $j \in \{1, 2\}$.

Map-Reduce in Query Processing (17)

A Worked-Out Example (6)

- The reducer implements the following query:

$$V \leftarrow_{dept\_name} \gamma_{sum(salary)}(B)$$

- The result produced by each reducer instance running in parallel is a (single tuple) horizontal fragment of the final result of the query, so to obtain the latter we use an iterated union:

$$\bigcup_{k \in dom(dept\_name)} V_k$$

Summary

21st Century DBMSs: Massively-Distributed Data Processing (2)

- Massively-distributed processing over commodity clusters has proved itself in settings where the tasks that are parallelized have constrained expressiveness.
- Currently, it remains an open question whether a map-reduce engine of the kind used by Google and Yahoo can be used for more than analytical workloads.

Section 2

Map-Reduce Semantics for Relational-Algebraic Operators

Not All Queries Go Through Masses of Data

Even if the Database is Massive Alright

- Database queries operate on large-scale data.
- However, although the database may be large, many queries only retrieve small amounts of data, because, among other reasons, optimizers are so adept (as we have seen) at minimizing I/O.
- A paradigmatic query might ask for the balance of one particular bank account.
- Such queries do not constitute good applications of map-reduce engines.

Some Queries Do Go Through Masses of Data

Even if the Database is Not That Massive

- A class of queries for which applying map-reduce engines seems well motivated comprises those in extract-transform-load (ETL) systems for data warehousing.
- Briefly recall that, in this context, ETL systems take all data out of an operations DB (i.e., with a normalized schema for update-intensive OLTP workloads) and generates a multidimensional data warehouse (by denormalization, which involves many complex joins, and partition-based aggregation, also expensive) for OLAP.
- Even for relatively small operations DB this process can give rise to very processing-intensice queries.

RA Operations in Map-Reduce (1)

We will cover:

- selection
- projection
- union, intersection, difference
- natural join
- partitioned (or group-by) aggregation

RA Operations in Map-Reduce (2)

Selection

- Selection is captured by the mapper alone, the reducer has not work to do: it's the identify function.
- For $\sigma_C(R)$, where R is the single input relation and C is a predicate on the columns of R, we define:

  map: For each t in the input, if $C(t)$ is true, then emit the key-value pair (t, t), i.e., set both key and value to t.

  reduce: For each (t, t) from any of many mappers, emit (t, t).

- Note that the output is not a relation, but a relation is easily obtained may taking only either the key or the value from the pair emitted by the reducer.

RA Operations in Map-Reduce (3)

Projection

- Under set semantics, projection must eliminate duplicates.
- For $\pi_A(R)$, where R is the single input relation and A is the set of columns in R to be kept, we define:

  map For each t in the input, construct from t a tuple t' containing only the columns in A and emit the key-value pair (t', t'), i.e., set both key and value to t'.

  reduce For each key t' from any of many mappers, there will be one or more (t', t') pairs (i.e., the reducer takes pairs of the form $(t', [t', t', \ldots, t'])$, emit exactly one pair (t', t') for each key t'.

- Because all the reducer does is eliminate duplicates, if each instance of the mapper eliminates duplicates locally, then the semantics is preserved. Still, a reducer phase is needed for duplicates from different mapper instances.
- Again, result type coercion would be needed.

RA Operations in Map-Reduce (4)

Union

- Union has a merely formatting mapper and a duplicate-eliminating reducer.
- Because of the schema compatibility requirement, it doesn't matter that mappers get assigned chunks from either input relation.
- For $R \cup S$, where R and S are the input relations, we define:
- map: For each t in the input, emit the key-value pair (t, t), i.e., set both key and value to t.
  reduce: For each key t from any of many mappers, there will be one or two (t, t) pairs (i.e., the reducer takes pairs of the form $(t, [t, t])$, emit exactly one pair (t, t) for each key t.
- The same observations made for projection hold here too.

RA Operations in Map-Reduce (5)

Intersection

- The mapper for union can be used for intersection too.
- However, the reducer must only emit a pair if the corresponding tuple occurs in both relations.
- For $R \cap S$, where R and S are the input relations, we define:
- map For each t in the input, emit the key-value pair (t, t), i.e., set both key and value to t.
  reduce For each key t from any of many mappers, there will be one or two (t, t) pairs (i.e., the reducer takes pairs of the form $(t, [t, t])$. If the second element is a singleton, emit nothing, otherwise emit exactly one pair (t, t) for each key t.

RA Operations in Map-Reduce (6)

Difference

- Difference is more complex because (as is the case for all set operations) each mapper instance may have in its input a mixture of tuples, some from one input relation and some from the other.
- The information as to which relation the tuple is from needs to be passed on to the reducer.
- For $R \setminus S$, where R and S are the input relations, we define:
- map For each t in the input, if $t \in R$, emit the key-value pair $(t, 'R')$, otherwise emit the key-value pair $(t, 'S')$, where the second element may be implemented economically with a single bit to indicate membership in either of the two input relations.

  reduce For each key t from any of many mappers, if the associated value list is $['R']$, then emit exactly one pair (t, t) for each key t, otherwise emit nothing.

RA Operations in Map-Reduce (7)

Natural Join (1)

- To see how to implement natural join, consider the concrete example of $R \bowtie S$ where the schema of R is (a, b) and that of S is (b, c).
- We must find tuples that agree on the value for b, i.e., the second column of R and the first column of S.
- We use the b-value in each tuple (taking into account which column of either relation is the b-column) as key and the rest of the tuple as value.
- The information as to which relation the tuple is from is therefore passed on to the reducer.

RA Operations in Map-Reduce (8)

Natural Join (2)

- For $R \bowtie S$, where R and S are the input relations with schemas (A, B) and (B, C), resp., we define:
- map For each t in the input, if $t \in R$, emit the key-value pair $(b, (\text{'R'}, a)$, otherwise emit the key-value pair $(b, (\text{'S'}, c)$, where a, b, c are values in the columns A, B, C, resp..

  reduce For each key b from any of many mappers, the associated value list will contain pairs of the form $(\text{'R'}, a)$ or $(\text{'S'}, c)$. Then, compute

```
for each key b
    list = []
    for each ('R', a)
        for each ('S', c)
            list.append((a,b,c))
    emit(b, list)
```

- Note that the key emitted by the reducer is irrelevant, and that the natural join does emit a relation.

RA Operations in Map-Reduce (9)

Natural Join (3)

- Although this was defined for two binary relations, it generalizes.
- We can think of A in R as representing all the columns in the schema of R and not in the schema of S, of B as all the common columns between R and S, and of C as all the columns in the schema of S and not in the schema of R.
- If $schema(.)$ returns the set of columns in the table passed as argument, then
  - The key for a tuple in R or S is $schema(R) \cap schema(S)$.
  - The value for a tuple in R is $schema(R) \setminus schema(S)$.
  - The value for a tuple in S is $schema(S) \setminus schema(R)$.

RA Operations in Map-Reduce (10)

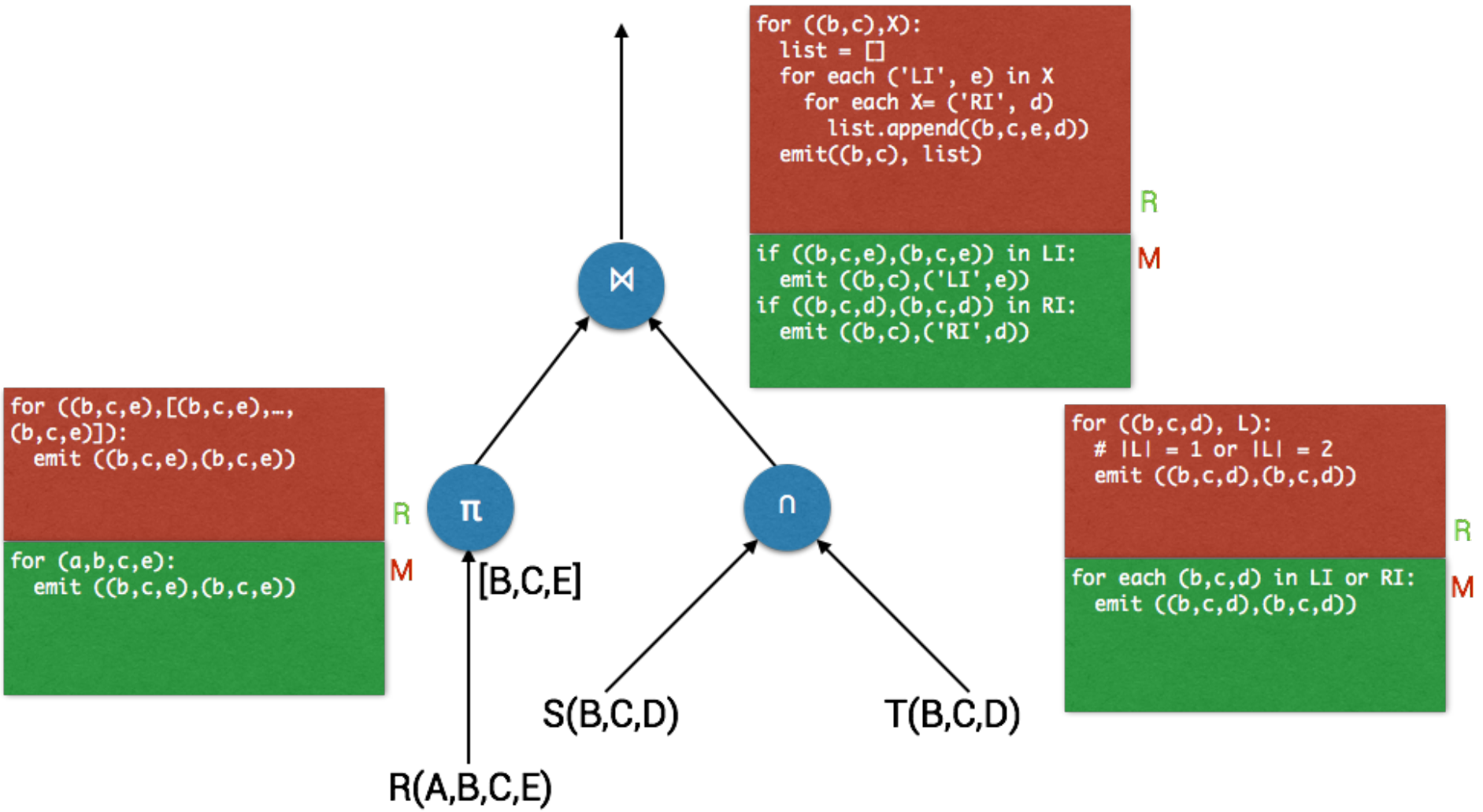
Partitioned (Group-By) Aggregation

- As with the natural join above, we use the minimal case for clarity of exposition by assuming there is a single grouping attribute and a single aggregation function. (Again, it generalizes).
- Consider the concrete example of a relation $R(A, B, C)$ to which we apply the operation ${}_A\gamma_{\phi(B)}$ where ϕ is an aggregation function, B the column being aggregated, and A the grouping column.
- map For each tuple (a, b, c) in the input, emit the key-value pair (a, b), i.e., the grouping attribute is the obvious key and, under map-reduce semantics, the desired partition of the input follows from that.

  reduce For each key a from any of many mappers, there will be one or more (a, b) pairs (i.e., the reducer takes pairs of the form $(a, [b_1, b_2, \ldots, b_n])$, emit exactly one pair (a, x) for each key a, where $x = \phi([b_1, b_2, \ldots, b_n])$ (e.g., if ϕ is `SUM`, then $x = b_1 + b_2 + \ldots + b_n$).

RA Operations in Map-Reduce (11)

An Example Query Plan



This Week: Summary

- We have looked into querying (live, distributed) data on the web, as opposed to querying (locally-materialized) data from/for the web.
- We have also briefly looked into the NoSQL approach to data management.
- Finally, we have delved a little on map-reduce engines as a platform for query processing.

This Week: Learning Objectives

- We have aimed to acquire a deeper understanding of distributed querying on the web.
- We have aimed to acquire a deeper understanding of distributed stores under the NoSQL paradigm.
- We have aimed to gain an insight into how map-reduce engines offer an opportunity for massively-parallelization of query execution.

Thanks, and Best Wishes!

- Thank you for taking this course unit!
- We hope that it has given you a basic start into exploring the significant challenges involved in querying data on the Web.
- If you are interested in doing a PhD in any of the areas touched here, we would be happy to help you focus your thoughts if you wish to discuss it with us.
- Whatever you go on to do, we wish you all the best!

Part VI

Supplementary Material on Massively-Parallel Schemes

Supplementary Material: Outline

8 Data Appliances
- Scale-Up v. Scale-Out
- Massively-Parallel Data Appliances
- An Example MPP Data Appliance
- Data Appliances v. Cluster-Based Schemes

Supplement 3

Data Appliances

Section 1

Scale-Up v. Scale-Out

Massive Parallelism and Distribution (1)

Tera- to Petabyte-Scale Data Volumes

- Recall that the post-1990s success of parallel/distributed architectures for DBMSs was attributed to three main factors, viz.:
  - the ascendancy of the relational model,
  - the high-quality of commodity hardware and network components, and
  - the enabling power of modern software mechanisms.
- All of these advances have continued into the 2000s and the latter two in particular underpin the cutting edge of massively-parallel architectures that are likely to dominate the future of tera- to petabyte-scale data management and beyond, viz.:
  - **massively-parallel data appliances**
  - **massively-distributed cluster schemes**

Massive Parallelism and Distribution (2)

Data Appliances

- Massively-parallel data appliances can be seen as scale-up (or vertically-scaled) designs.
- They pack software and hardware in a box, deploying preconfigured and tuned DBMSs onto preconfigured and tuned processors and disks wired by a preconfigured and tuned interconnect.
- They adopt optimization schemes based on special storage strategies to minimize reliance on indexes and maximize the use of highly-parallel, high-performance scans and hash joins in query plans.
- While they major on OLAP-style queries, they are SQL engines.
- Consistently with the read-only nature of their typical workload, some data appliances remain shared-disk designs.

Massive Parallelism and Distribution (3)

Cluster-Based Schemes

- Massively-parallel cluster-based schemes replace scale-up (bigger boxes) with scale-out (more boxes) for their design strategy.
- They are the latest realization of extreme shared-nothing with the twist of custom-built software infrastructure for merging and splitting as well as scheduling and fault-tolerance.
- While they major in un- and semi-structured data processing, they can be targets for SQL-based query workloads.

Section 2

Massively-Parallel Data Appliances

Data Appliances (1)

A Response to One of the Critiques of Classical DBMSs

- In order to keep TCO down, administration and operation costs need to be kept down.
- The notion of an **appliance** connotes precisely this works-out-of-the-box, plug-and-run ideal.
- The notion first emerged in network management, then was picked up in keyword-based search applications.
- An appliance can be seen as special-purpose hardware running special-purpose software for a well-defined class of computing tasks.
- Both the hardware and the software are preconfigured and tuned (and this is where appliance vendors compete and aim to make their money).

Data Appliances (2)

Major Characteristics (1)

- **Shared-nothing massively-parallel processing (MPP)**: In order to deliver performance, data appliances make heavy use of all the advances in parallel DBMS we have studied.
- **Simpler QEPs on more raw computing power**: In order to both avoid high administration and operation costs and improve raw performance, data appliances strongly favour a simpler space of QEPs and physical schemas and pack enough high-quality hardware and fine-tuned software to make them go faster than if more optimized, more complex QEPs were used.

Data Appliances (3)

Major Characteristics (2)

- QEP simplification is founded on two characteristics:
  - **Index-light**: avoid indices which are costly to design, store, maintain and make use of without the highest-quality DBAs being in constant interaction with application designers.
  - **Scan-heavy**: avoid random disk accesses and ensure instead that sequential scans come close to potential speed-up by physical designs that are tuned to high-quality, specially-configured, fine-tuned hardware.
- The major technical challenges centre around balanced, efficient data movements, i.e., the dynamic management of intermediate result sets and their timely, fast delivery to capable, fast-performing nodes.

Data Appliances (4)

Major Types/Current Players

One classification of data appliances is as follows:

Type 0 appliances use customized hardware. They are distinctive in using custom chips (or field-programmable gate arrays) and aggressively push computation to them in order to obtain better performance.

Type 1 appliances are custom assemblies of standard hardware that do not make use of chip-level tuning.

Type 2 appliances preconfigure and tune standard software and hardware into a coherent, tightly-coupled unit component.

Examples include:

Type 0 XtremeData, Netezza (now IBM)

quasi-**Type 1** Teradata, Greenplum

Type 2 DATAllegro (now Microsoft), Oracle

Data Appliances (5)

Raw Computing Power Needed

Data appliances rely on:

- Many, high-performance, high-capacity components, such as:
  - multiple top-end processors with excellent caching performance in the presence of massive amounts of RAM,
  - multiple top-end storage units, with redundancy for reliability and for high-degree of parallelism, and
  - top-end interconnect to avoid slow down when in transit;
- Balancing strategies to extract the best performance of such components.

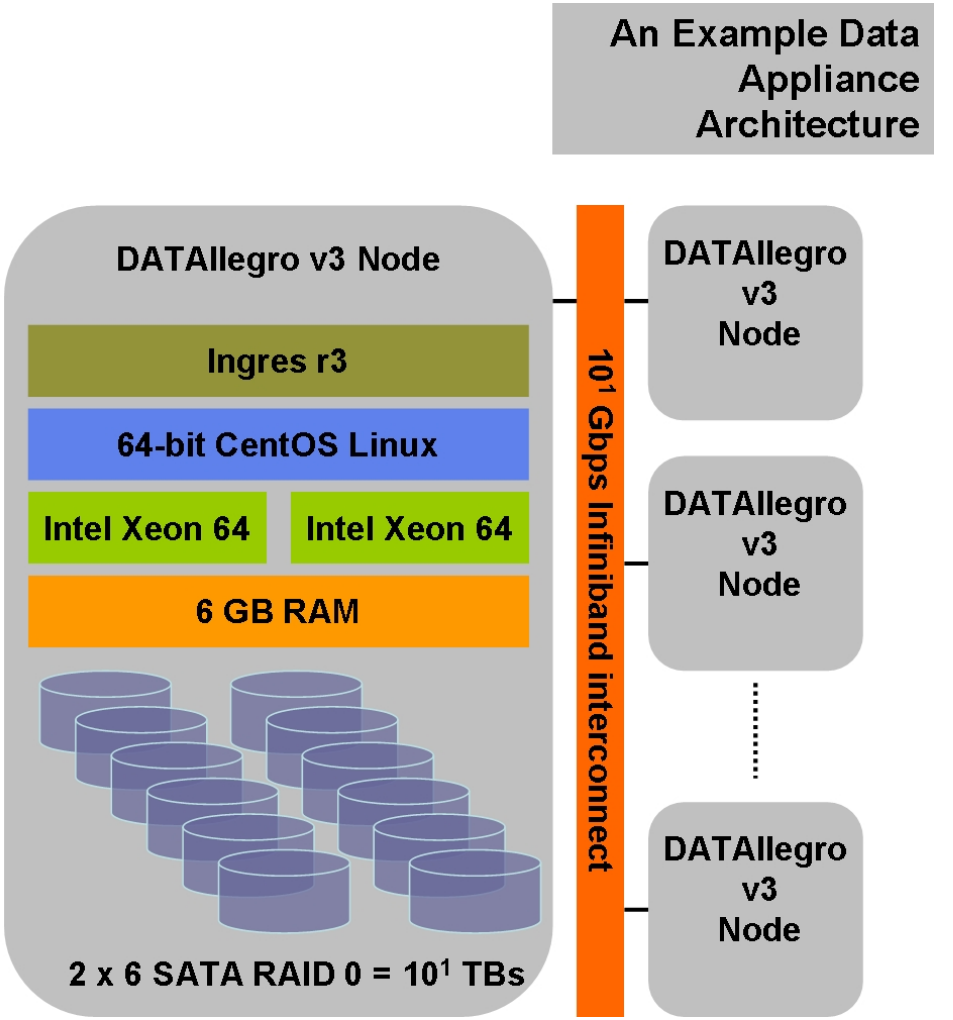
Section 3

An Example MPP Data Appliance

Data Appliances (6)

2007 DATAllegro v3: A Type 2 Data Appliance (1)

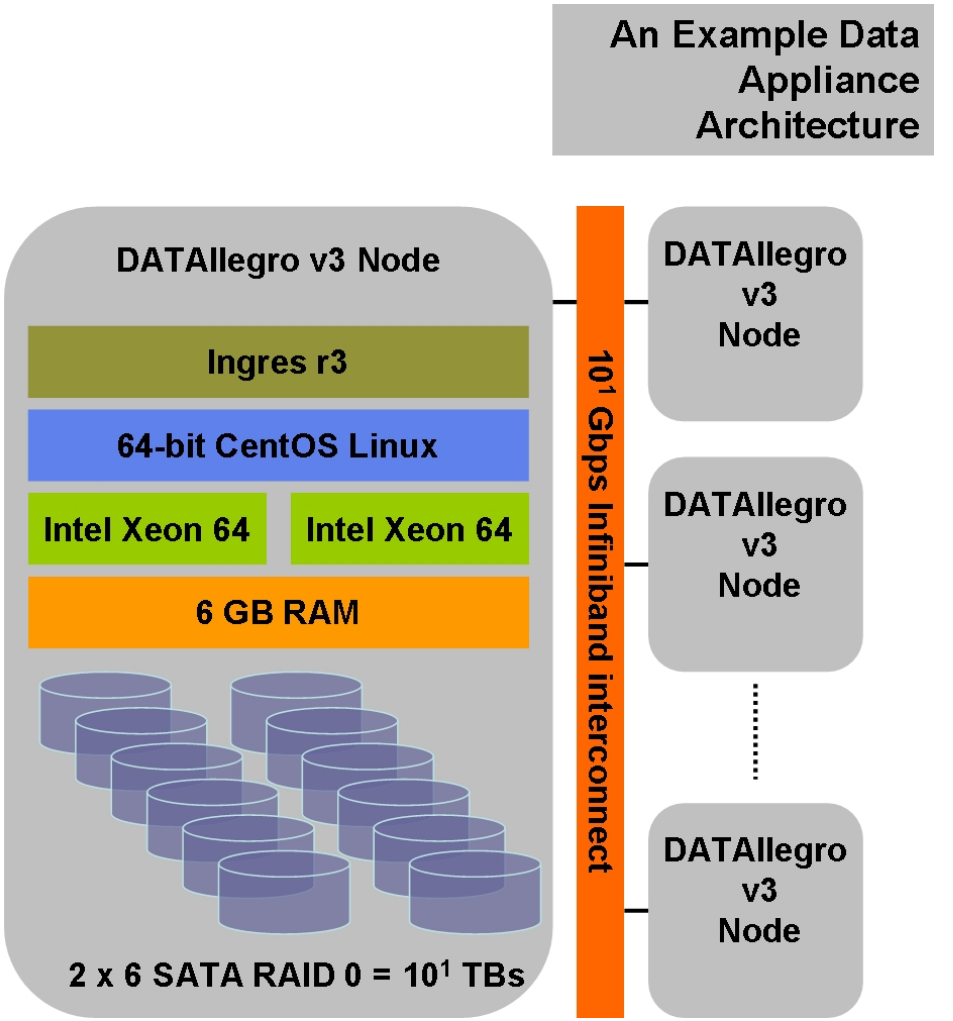
- An appliance comprises multiple nodes, in a one-master, many-slaves relationship.
- The master node is responsible for query optimization, parallelization, concurrency, etc.
- Slave nodes are responsible for storing data and executing the QEPs sent by the master node.



Data Appliances (7)

2007 DATAllegro v3: A Type 2 Data Appliance (2)

- Each node runs an open source DBMS that is highly-, finely-tuned for the required query workloads (OLAP in the case of DATAllegro).
- Storage used a RAID (Redundant Array of Independent Drives) design intent on maximizing read performance.
- It uses top-end commodity components for each node and interconnects them with top-end network.
- If a node fails, another node picks up the role and load until it is slid out and a replacement is slid in.



Data Appliances (8)

Scan-Heavy Strategy

- As data is loaded, it is automatically hash-partitioned across the slave nodes.
- In addition, range- and hash-partitioning are used within a node to capitalize on the RAID storage.
- Tables can be replicated and/or partitioned.
- The query optimizer makes the decision between:
  - running the query within a node using replicas
  - running the query across nodes using partitioning
- The query optimizer is also fine-tuned to make it possible to use sequential scans and hash joins.
- Transfer units of storage are very large (e.g., 24-MB blocks).

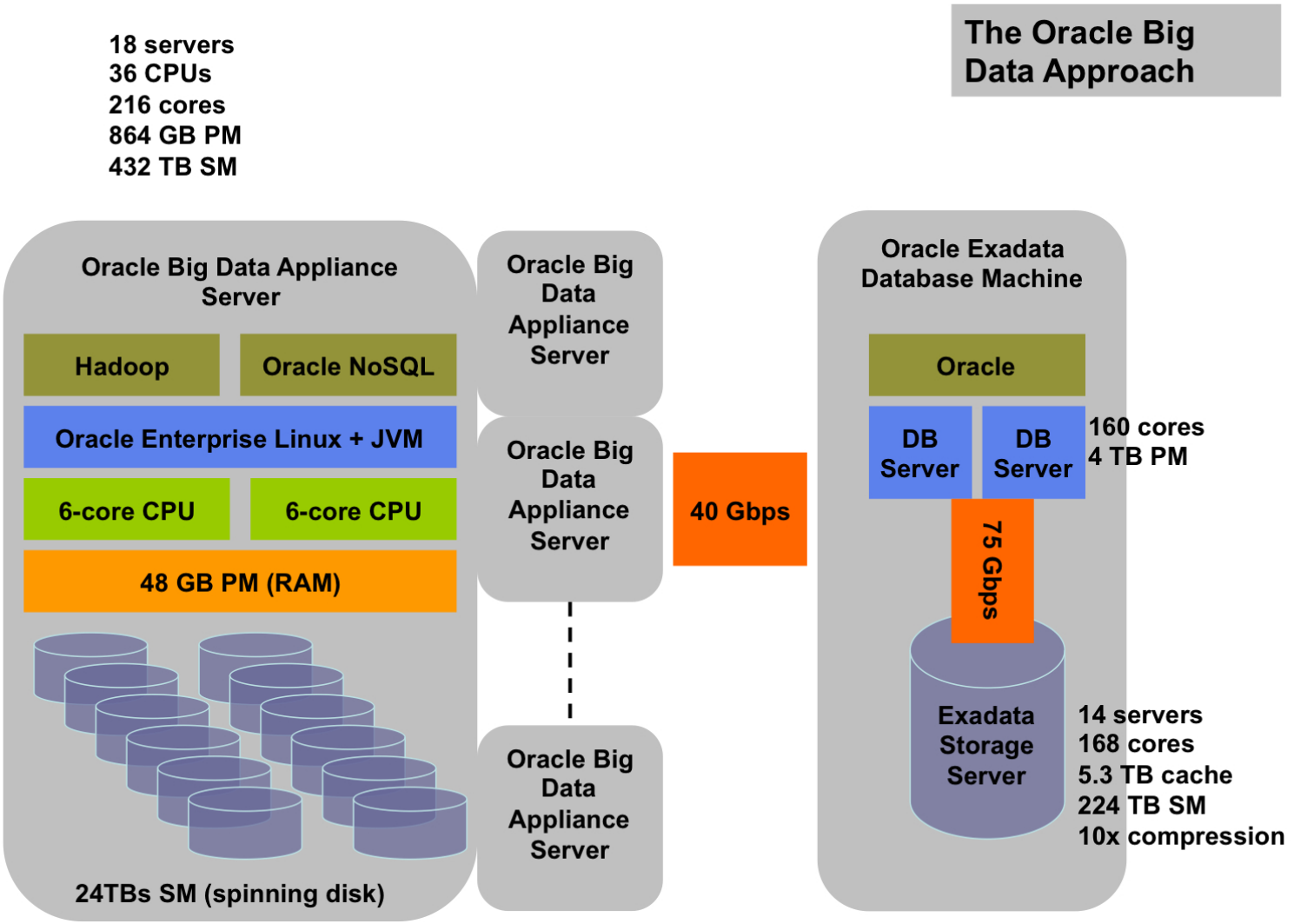
Data Appliances (9)

Index-Light Strategy

- The index-light approach means that often one order-of-magnitude less storage space is needed, even without compression.
- The index-light approach implies not just less hardware but also less administration (to decide, design, maintain and make use of indices) and more consistent performance at a lower price/performance ratio.
- Rather than being slower at finding exactly the true positives, use raw power to read and process even the false positives faster.
- Indices can still be used: most data appliances are (or contain) a fully-fledged DBMS.

Data Appliances (10)

2011 The Oracle Big Data Approach = US$ 0.55M + US$ 2M



Summary

21st Century DBMSs: Massively-Parallel Data Processing

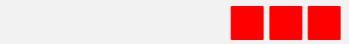
- A new breed of data management solutions has emerged recently that is a response to the need to lower TCO even in the presence of massive volumes of data and complex query workloads.
- The notion of a computing appliance first emerged in network management, then guided the development of integrated hardware/software products in the enterprise search market and is now being used to underpin cutting-edge solutions for OLAP querying.
- Data appliances use massively-parallel shared-nothing architectures for performance.
- They configure and tune the software and hardware to perform well with much reduced administration overheads.
- This is achieved by a combination of raw power and simplified, index-light, scan-heavy QEPs.

Section 4

Data Appliances v. Cluster-Based Schemes

Data Appliances v. Cluster-Based Schemes (1)

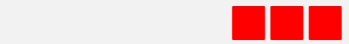
TCO, Again



- Data appliances are perceived as having succeeded in reducing the administration and operation components of TCO, but their acquisition costs are in the order of tens of thousands per terabyte.
- Even though, by definition, appliances are designed, sold and deployed as an integrated hardware/software solution, there are two components to their acquisition cost:
  - the hardware, i.e., the top-end components that combine to produce the raw computing power without which the simplified QEPs would not deliver performance, and
  - the software, i.e., the partitioning and scheduling components that ensure that computing power is made the most use of and that constitute the competitive ground for vendors.

Data Appliances v. Cluster-Based Schemes (2)

Major Types



Recall:

Type 0 appliances use customized hardware. They are distinctive in using custom chips (or field-programmable gate arrays) and aggressively push computation to them in order to obtain better performance.

Type 1 appliances are custom assemblies of standard hardware that do not make use of chip-level tuning.

Type 2 appliances preconfigure and tune standard software and hardware into a coherent, tightly-coupled unit component.

Data Appliances v. Cluster-Based Schemes (3)

Clusters as "Type 3" Appliances

- Cluster-based approaches can be seen as an attempt to move onwards from Type 2 data appliances just as Type 2 appliances moved from Type 1 and Type 0 ones, i.e., in the direction of non-proprietary technologies.
- Thus, cluster-based schemes
  - can deliver performance even without specially configured and tuned assemblies of top-end hardware and network components, and
  - use open-source partitioning and scheduling software.
- In this way, cluster-based schemes promise to lower the acquisition component of TCO and not just the administration and operation ones.
- While data appliances could be argued to be scale-up designs, cluster-based schemes are scale-out designs.

Acknowledgements (1)

The material presented mixes original material by the author as well as material adapted from

- [Monash, 2007a, Monash, 2007b]

The author gratefully acknowledges the work of the author cited while assuming complete responsibility any for mistake introduced in the adaptation of the material.

References I


Alberton, L. (2011).
NoSQL databases: Why, what and when.
Talk at UK PHP Conference.

`http://www.slideshare.net/quipo/nosql-databases-why-what-and-when`.


Baker, J., Bond, C., Corbett, J., Furman, J. J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A., and Yushprakh, V. (2011).
Megastore: Providing scalable, highly available storage for interactive services.
In CIDR, pages 223–234.
http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf.

References II

Brewer, E. A. (2000).
Towards robust distributed systems (abstract).
In PODC, page 7.
http://doi.acm.org/10.1145/343477.343502.

Cattell, R. (2010).
Scalable sql and nosql data stores.
SIGMOD Record, 39(4):12–27.
http://doi.acm.org/10.1145/1978915.1978919.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008).
Bigtable: A distributed storage system for structured data.
ACM Trans. Comput. Syst., 26(2).
http://doi.acm.org/10.1145/1365815.1365816.

References III

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007).
Dynamo: amazon's highly available key-value store.
In SOSP, pages 205–220.
http://doi.acm.org/10.1145/1294261.1294281.

Gilbert, S. and Lynch, N. A. (2002).
Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services.
SIGACT News, 33(2):51–59.
http://doi.acm.org/10.1145/564585.564601.

References IV


Harizopoulos, S., Abadi, D. J., Madden, S., and Stonebraker, M. (2008).
Oltp through the looking glass, and what we found there.
In SIGMOD Conference, pages 981–992.
http://doi.acm.org/10.1145/1376616.1376713.


Leskovec, J., Rajaraman, A., and Ullman, J. (2015).
Mining of Massive Datasets.
Stanford University.
http://www.mmds.org/.


Monash, C. A. (2007a).
Design choices in mpp data warehousing.
Technical report, Monash Research White Paper.
http://www.monash.com/DATAllegro-V3.pdf.

References V

Monash, C. A. (2007b).
Index-light mpp data warehousing.
Technical report, Monash Research White Paper.
http://www.monash.com/MPP-Appliance.pdf.

Stonebraker, M. and Cattell, R. (2011).
10 rules for scalable performance in 'simple operation' datastores.
Commun. ACM, 54(6):72–80.
http://doi.acm.org/10.1145/1953122.1953144.