CS 348: Lecture 20

Acknowledgements & Beyond CS 348

Semih Salihoğlu

March 29, 2022

UNIVERSITY OF WATERLOO | DSG Data Systems Group

Announcements

- A5 due on April 1st midnight
- A6 will be released on April 4th midnight (you have 24 hrs)
  - Cumulative
  - Expect Relational Algebra Question
  - Expect SQL Question
  - Expect a Processing or Optimization Question
  - Expect a Transaction Question
  - But other topics will be covered too
  - Length will be fair

Outline

- Courses/Topics Beyond 348
- GraphflowDB: Graph Database Management Systems
- MapReduce

Outline

- Courses/Topics Beyond 348
- GraphflowDB: Graph Database Management Systems
- MapReduce

CS 348 Diagram

User/Administrator Perspective

Primary Database Management System Features

- Data Model: Relational Model
- High Level Query Language: Relational Algebra & SQL
- Integrity Constraints
- Indexes/Views
- Transactions

Relational Database Design

- E/R Models
- Normal Forms

How To Program A DBMS (0.5-1 lecture)

- Embedded vs Dynamic SQL
- Frameworks

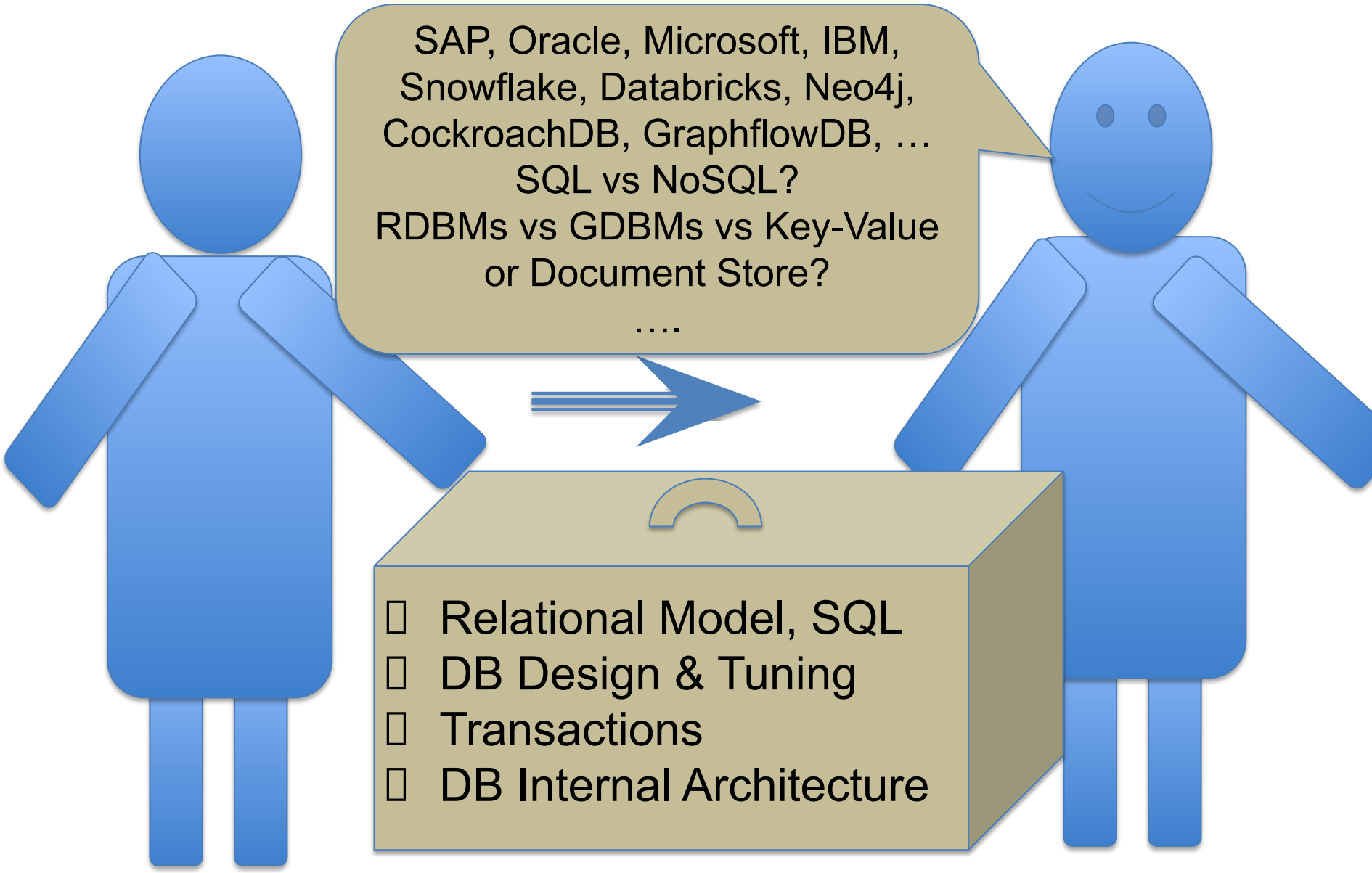
DBMS Architect/Implementer Perspective

- Physical Record Design
- Query Planning and Optimization
- Indexes
- Transactions

Other (Last 1/2 Lectures)

- Graph DBMSs
- MapReduce: Distributed Data Processing Systems

Before/After CS 348



Beyond CS 348

- CS 448: Database Systems Implementation
  - Design and implementation of core DBMS components:
    - Data layout, disk-based data structures, indexing, query processing algorithms, query optimization, transactional concurrency control, logging and recovery.
- CS 451: Data-Intensive Distributed Computing
  - How to scale data-intensive computations (e.g., petabyte scale)
  - Big data stack, e.g., Spark, Storm, Hadoop
  - Text, graphs, data mining, machine learning systems software
- Interested in research on data processing and management systems?
  - Reach out to me and DSG faculty for research positions.

Outline

- Courses/Topics Beyond 348
- GraphflowDB: Graph Database Management Systems
- MapReduce

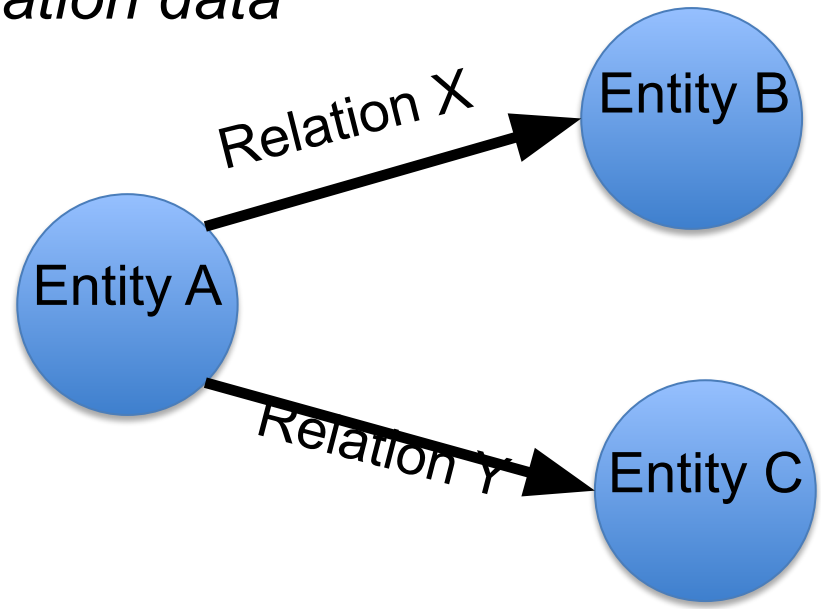
Graphs vs Tables?
Never Ending Dichotomy in Data Models

Two of the most natural data structures to model enterprise application data

Entity A		
…	…	…
…	…	…
…	…	…
…	…	…

Entity B	
…	…
…	…
…	…
…	…

Relation X		
…	…	…
…	…	…
…	…	…
…	…	…



- Other data structures
  - Arrays/Matrices: images, videos, locations, etc.
  - Documents
  - Lists (earlier)

Key Takeaway: Dual View of Enterprise Data

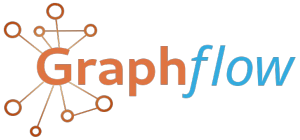
- Graphs, tables, or arrays don't exist in real world
- Real events/facts that enterprises store as records

More interesting question: When is one model more appropriate than another?

Graph Processing Systems

Graph DB Management Systems

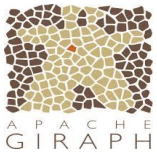
Amazon Neptune



RDF Systems

Amazon Neptune

Graph Analytics Systems



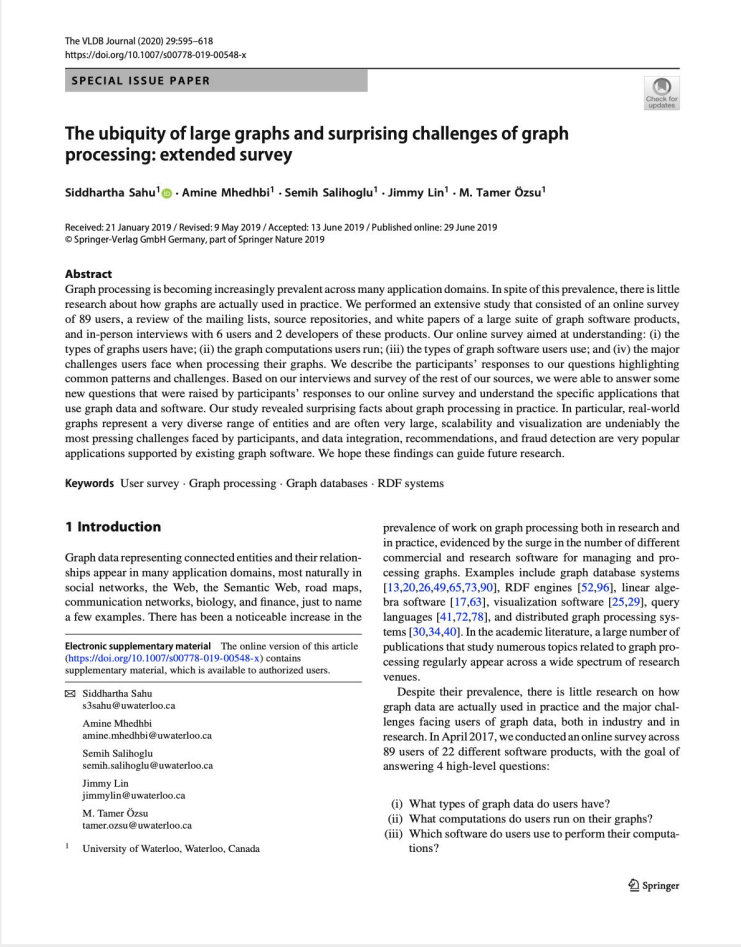
Graph Visualization Software

GraphWrangler

How Are Graphs and Graph Systems Used In Practice?

A User Survey

Sahu et. al. VLDBJ 2019



The VLDB Journal (2020) 29:595–618
https://doi.org/10.1007/s00778-019-00548-x

SPECIAL ISSUE PAPER

The ubiquity of large graphs and surprising challenges of graph processing: extended survey

Siddhartha Sahu[1] · Amine Mhedhbi[1] · Semih Salihoglu[1] · Jimmy Lin[1] · M. Tamer Özsu[1]

Received: 21 January 2019 / Revised: 9 May 2019 / Accepted: 13 June 2019 / Published online: 29 June 2019

Abstract
Graph processing is becoming increasingly prevalent across many application domains. In spite of this prevalence, there is little research about how graphs are actually used in practice. We performed an extensive study that consisted of an online survey of 89 users, a review of the mailing lists, source repositories, and white papers of a large suite of graph software products, and in-person interviews with 6 users and 2 developers of these products. Our online survey aimed at understanding: (i) the types of graphs users have; (ii) the graph computations users run; (iii) the types of graph software users use; and (iv) the major challenges users face when processing their graphs. We describe the participants' responses to our questions highlighting common patterns and challenges. Based on our interviews and survey of the rest of our sources, we were able to answer some new questions that were raised by participants' responses to our online survey and understand the specific applications that use graph data and software. Our study revealed surprising facts about graph processing in practice. In particular, real-world graphs represent a very diverse range of entities and are often very large, scalability and visualization are undeniably the most pressing challenges faced by participants, and data integration, recommendations, and fraud detection are very popular applications supported by existing graph software. We hope these findings can guide future research.

Keywords User survey · Graph processing · Graph databases · RDF systems

1 Introduction

Graph data representing connected entities and their relationships appear in many application domains, most naturally in social networks, the Web, the Semantic Web, road maps, communication networks, biology, and finance, just to name a few examples. There has been a noticeable increase in the prevalence of work on graph processing both in research and in practice, evidenced by the surge in the number of different commercial and research software for managing and processing graphs. Examples include graph database systems [13,20,26,49,65,73,90], RDF engines [52,96], linear algebra software [17,63], visualization software [25,29], query languages [41,72,78], and distributed graph processing systems [30,34,40]. In the academic literature, a large number of publications that study numerous topics related to graph processing regularly appear across a wide spectrum of research venues.

Despite their prevalence, there is little research on how graph data are actually used in practice and the major challenges facing users of graph data, both in industry and in research. In April 2017, we conducted an online survey across 89 users of 22 different software products, with the goal of answering 4 high-level questions:

(i) What types of graph data do users have?
(ii) What computations do users run on their graphs?
(iii) Which software do users use to perform their computations?

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s00778-019-00548-x) contains supplementary material, which is available to authorized users.

✉ Siddhartha Sahu
s3sahu@uwaterloo.ca
Amine Mhedhbi
amine.mhedhbi@uwaterloo.ca
Semih Salihoglu
semih.salihoglu@uwaterloo.ca
Jimmy Lin
jimmylin@uwaterloo.ca
M. Tamer Özsu
tamer.ozsu@uwaterloo.ca
[1] University of Waterloo, Waterloo, Canada

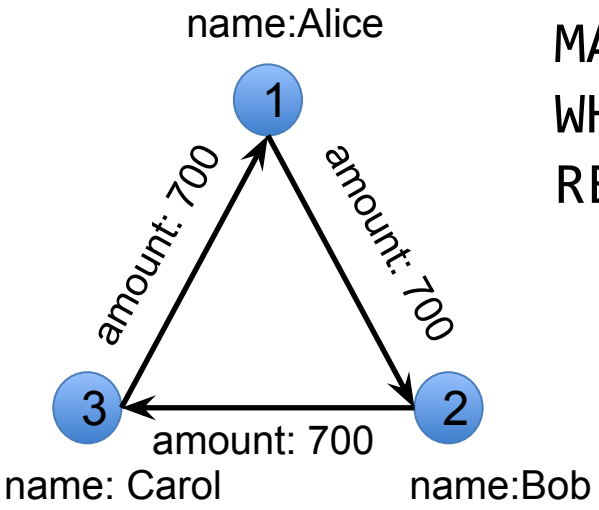
- Q1: Graph Data?
- Q2: Graph Computations?
- Q3: Graph Software?
- Q4: Main Challenges?
- Q5: Applications?

Graph Database Management Systems Overview

- Read-optimized database management systems designed for read-heavy workloads with large many-to-many (n-n) joins

Data Model

Labeled Graph



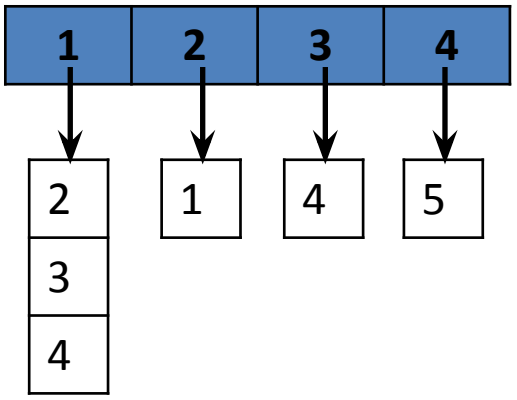
Query Language

Graph-specific SQL

```
MATCH a->b->c,c->a
WHERE a.age > 30
RETURN (a,b).amount
```

System

Storage: Graph-specific



Query Processor: Fast traversals, i.e., nested index loop joins

Differences Between Native GDBMSs vs RDBMSs (1)

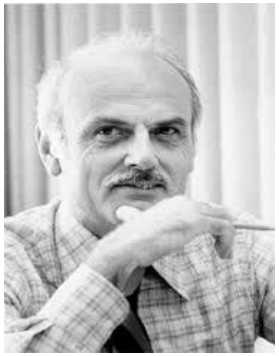
1. Pre-defined/Pointer-based joins vs Value-based Joins

Network Model (1960s)

IDS: First DBMS in history

Charles Bachman

Relational Model (1970s)



Ted Codd

Much of the derivability power of the relational algebra is obtained from the SELECT, PROJECT, and JOIN operators alone, provided the JOIN is not subject to any implementation restrictions having to do with predefinition of supporting physical access paths. A system has an *unrestricted join capability* if it allows joins to be taken wherein *any* pair of attributes may be matched, providing only that they are defined on the same domain

… but also the reason GDBMSs can be very fast at those joins.

Differences Between Native GDBMSs vs RDBMSs (2)

1. Pre-defined/Pointer-based joins (access paths)
2. Semi-structured data model
   - No fixed schema
3. Better support for variable-length recursive join queries

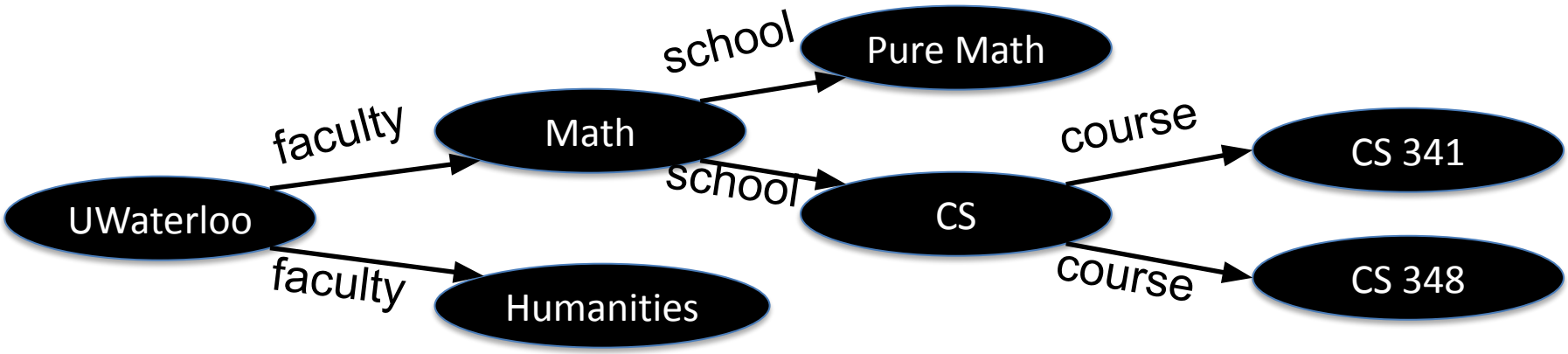
"Give me all direct or indirect possible sources of money flow into Alice's account from Canada."

```
MATCH a-[:Transfer*]->b
WHERE a.location=Canada AND b.owner=Alice
```

Can be done in recursive SQL but harder

Example Semi-structured Datasets (1)

- Application: Question Answering over knowledge bases: encyclopedic facts about real-world entities
  - Also called “Knowledge Graphs”
- Often too complex to structure: cannot have a fixed schema
- Consider encoding all facts about Canadian Universities
  - University of Waterloo vs York University
    - Different faculty structures
    - Different department structures
    - Different cirricula
- Cannot tabulate this information. Need a flexible model.



Links & a 1962 Drawing of IDS's Network Model

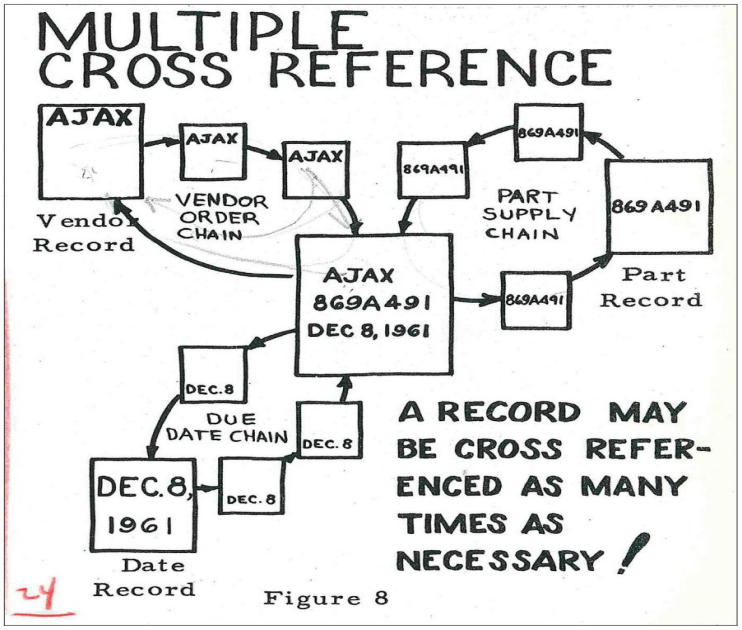


Figure 2. This drawing, from the 1962 presentation "IDS: The Information Processing Machine We Need," shows the use of chains to connect records. The programmer looped through GET NEXT commands to navigate between related records until an end-of-set condition is detected.

- Haigh's Paper on the Birth of DBs and IDS (2006): "A Veritable Bucket of Facts": Origins of the Data Base Management System
- Bachman's Turing Award Lecture (1973): Programmer As a Navigator
- Bachman's Talk (2002) at Computer History Museum
- Codd's Turing Award Lecture (1981): Relational Database: A Practical Foundation for Productivity

Graphflow

- In-memory Graph DBMS: Property graph model & openCypher
- Two primary features:

1. Very fast n-n joins:

GraphflowDB	RDBMS Literature
List-based, Factorized Query Processor	Factorized DBs
Multiway intersection-based joins	WCO joins
A+ Indexes: Flexible Adjacency List Indexes	Materialized Views

2. Scalability: compressed, in-memory columnar storage

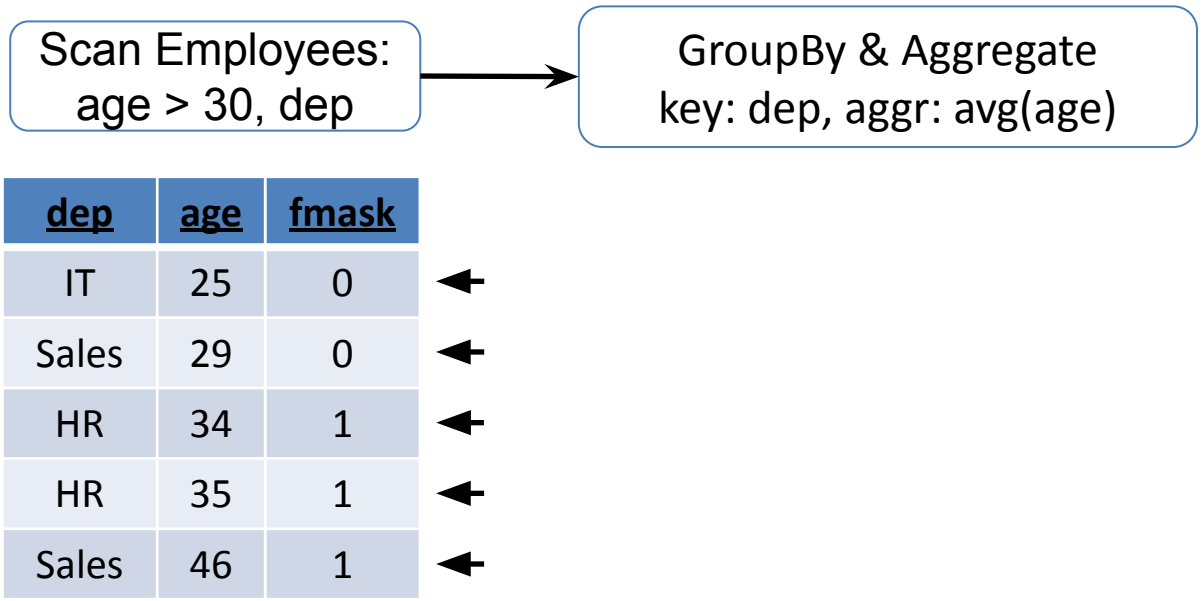
Flat Block-based Processing In Read-optimized RDBMSs

- Columnar RDBMSs, e.g., Vectorwise, MonetDB, Vertica
- Optimized for OLAP workloads (online analytical processing)
  - Read-heavy but joins are not n-n, and aggregation-heavy
- Operators process a *block of flat tuples at-a-time* in for loops.

```
SELECT dep, avg(age)
FROM Employee E
WHERE age > 30
```

Employee

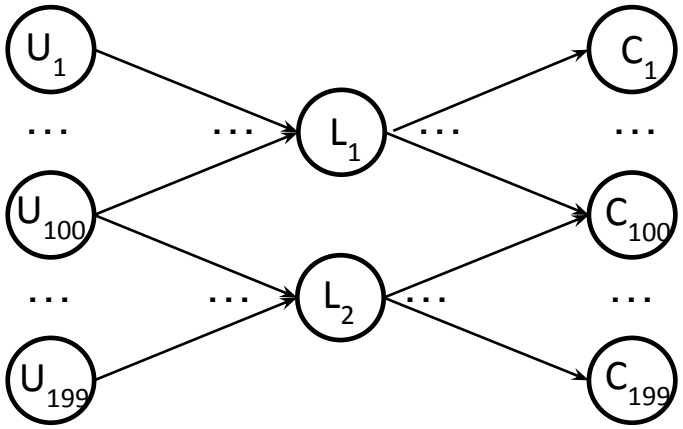
eID	age	dep
3121	25	IT
2431	29	Sales
1113	34	HR
5110	35	HR
9926	46	Sales

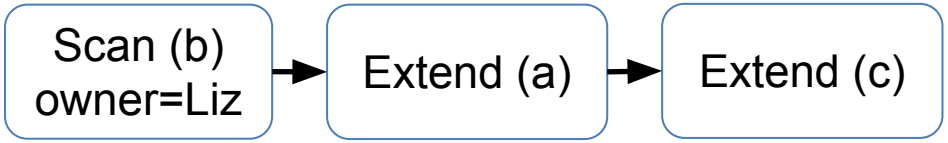


- Block-based: Good CPU utilization
- Flat Tuples: Not optimized for n-n joins

Value Repetitions When Performing n-n Joins

```
MATCH (a)->(b)->(c)
WHERE b.owner='Liz' AND
a.cntr='US' AND c.cntr='CAN'
```





b	owner	fmask
L_1	Liz	1
L_2	Liz	1
U_1	Ken	0
…	…	…
U_{199}	Bob	0

b	owner	a	c	fmask
L_1	Liz	U_1	C_1	1
L_1	Liz	U_1	C_2	1
	…	…	…	…
L_1	Liz	U_1	C_{100}	1
L_1	Liz	U_2	C_1	1
L_1	Liz	U_2	C_2	1
…	…	…	…	…
L_1	Liz	U_2	C_{100}	1

- Repetition happens b/c 1 group of vectors represent *a block of flat tuples*

F-Representation-based Processor (1)

Flat Representation

b	owner	a	c
L_1	Liz	U_1	C_1
	…	…	…
L_1	Liz	U_1	C_{100}
L_2	Liz	U_{100}	C_{100}
…	…	…	…
L_2	Liz	U_{199}	C_{100}

2x(100x100) tuples

F-Representation

b, owner		a		c
$\{L_1$, Liz$\}$	X	$\{U_1, \ldots, U_{100}\}$	X	$\{C_1, \ldots, C_{100}\}$
U				
$\{L_2$, Liz$\}$	X	$\{U_{100}, \ldots, U_{199}\}$	X	$\{C_{100}, \ldots, C_{199}\}$

2x(100+100) tuples

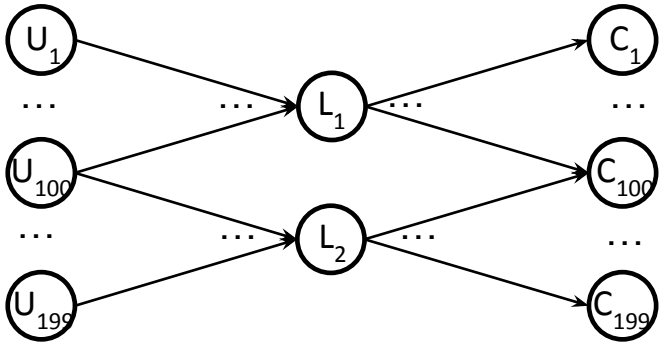
- Factorizing arbitrary relations is NP-hard
- But: outputs of a query Q can be factorized by analyzing *the conditional independence of the variables* in Q
  - E.g. in a → b → c, once b is fixed, a and c are independent

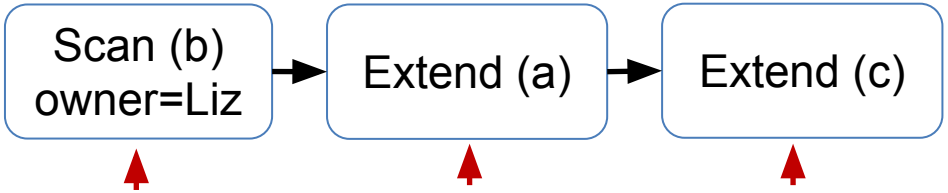
Multiple List Groups: Represent intermediate tuples in multiple vector groups

Olteanu et. al. Factorized Databases, SIGMOD Record, 2016

F-Representation-based Processor (2)

```
MATCH (a)->(b)->(c)
WHERE b.owner='Liz' AND
a.cntr='US' AND c.cntr='CAN'
```





List Group 1

b	owner	fmask
L_1	Liz	1
L_2	Liz	1
U_1	Ken	0
…	…	…
U_{199}	Bob	0

curIdx = 1

List Group 2

a	fmask
U_{100}	1
U_{101}	1
…	…
U_{199}	1

curIdx = -1

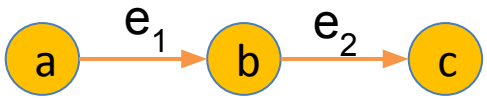
List Group 3

c	fmask
C_{100}	1
C_{101}	1
…	…
C_{199}	1

curIdx = -1

F-Representation-based Processor (3)

- Microbenchmark: 1-, 2-, 3-hop path queries
  - FILTER: last edge have a predicate, e.g.:

MATCH 

WHERE e2.date > 2021-03

  - COUNT: counts the number of paths
- LDBC Soc. Network Benchmark scale 100 (1.7B edges), Flickr (33M edges)
- GF-Flat vs GF-FRep

	LDBC100						FLICKR					
	FILTER			COUNT(*)			FILTER			COUNT(*)		
	1h	2h	3h	1h	2h	3h	1h	2h	3h	1h	2h	3h
Flat	0.02	1.4	40.0	0.02	0.3	6.9	0.03	1.3	14.8	0.03	0.5	4.1
FRep	0.01 **3.2x**	0.1 **12.7x**	6.9 **15.2x**	0.01 **3.2x**	0.02 **12.8x**	0.4 **19.4x**	0.01 **2.7x**	0.1 **13.7x**	1.2 **12.4x**	0.02 **2.1x**	0.02 **21.4x**	0.05 **80.6x**

runtimes in seconds

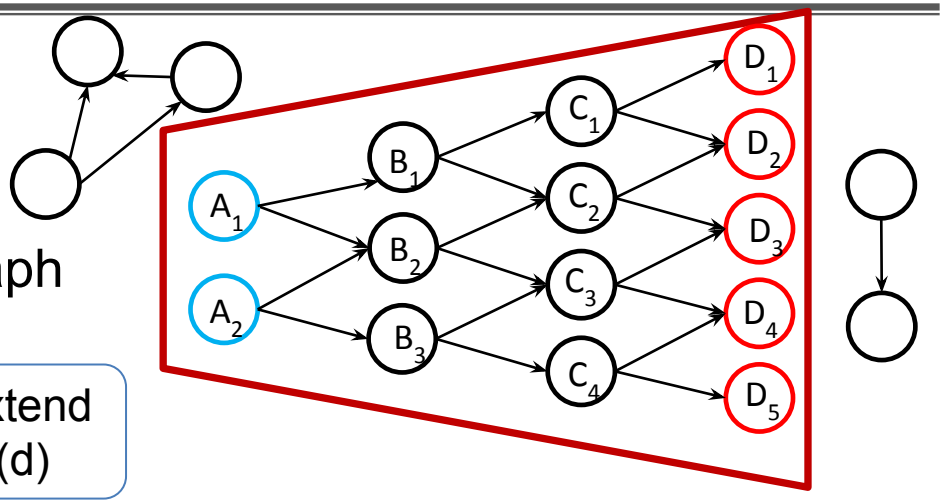
Also large differences with 3 baseline commercial systems (see paper)

Integrating Column-Oriented Storage and Query Proc. Techniques Into GDBMSs, Gupta et al., arXiv 2021

Further Value Repetitions

MATCH (a)→(b)→(c)→(d)

WHERE a.cntr='US' & e.cntr='CAN'



- Most compact output: relevant subgraph

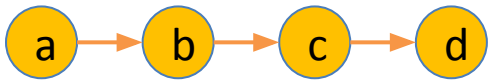
Scan (a) cntr=`US' → Extend (b) → Extend (c) → Extend (d)

LG1		LG2		LG3		LG4
a		**b**		**c**		**d**
…						
$\{A_1\}$	X	$\{B_1\}$	X	$\{C_2\}$	X	$\{D_2, D_3\}$
$\{A_1\}$	X	$\{B_2\}$	X	$\{C_2\}$	X	$\{D_2, D_3\}$
$\{A_1\}$	X	$\{B_2\}$	X	$\{C_3\}$	X	$\{D_3, D_4\}$
$\{A_2\}$	X	$\{B_2\}$	X	$\{C_2\}$	X	$\{D_3, D_4\}$
$\{A_2\}$	X	$\{B_2\}$	X	$\{C_3\}$	X	$\{D_3, D_4\}$
…						

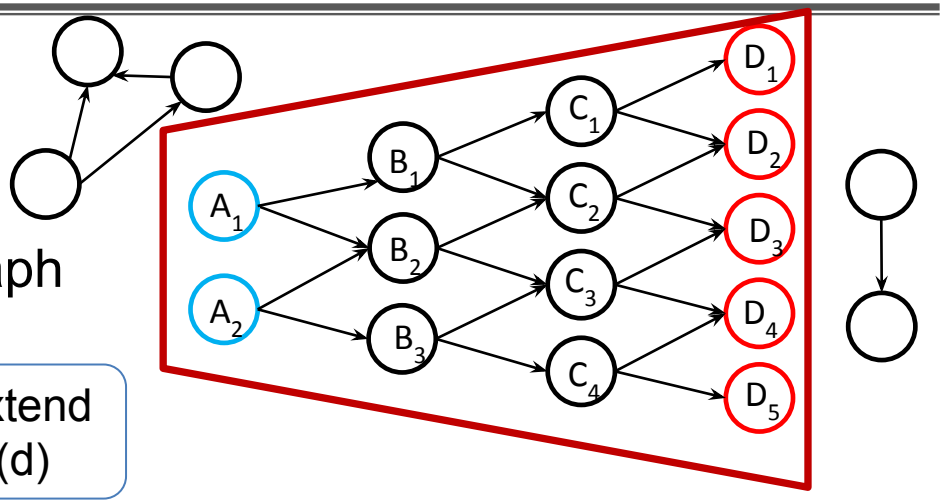
F-Representation

Further Value Repetitions

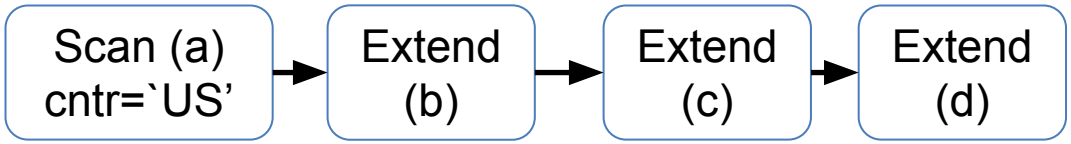
```
MATCH
```




```
WHERE a.cntr='US' & e.cntr='CAD'
```



- Most compact output: relevant subgraph



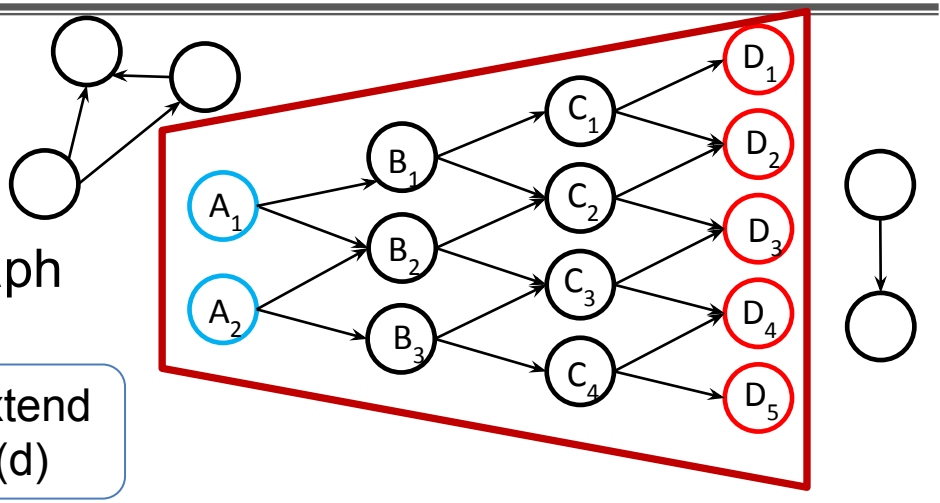
LG1		LG2		LG3		LG4
a		**b**		**c**		**d**
…						
$\{A_1\}$	X	$\{B_1\}$	X	$\{C_2\}$	X	$\{D_2, D_3\}$
$\{A_1\}$	X	$\{B_2\}$	X	$\{C_2\}$	X	$\{D_2, D_3\}$
$\{A_1\}$	X	$\{B_2\}$	X	$\{C_3\}$	X	$\{D_3, D_4\}$
$\{A_2\}$	X	$\{B_2\}$	X	$\{C_2\}$	X	$\{D_3, D_4\}$
$\{A_2\}$	X	$\{B_2\}$	X	$\{C_3\}$	X	$\{D_3, D_4\}$
…						

F-Representation

Further Value Repetitions

MATCH (a)→(b)→(c)→(d)

WHERE a.cntr='US' & e.cntr='CAD'



- Most compact output: relevant subgraph

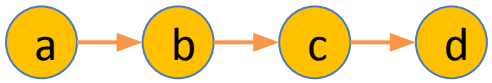
Scan (a) cntr=`US' → Extend (b) → Extend (c) → Extend (d)

LG1		LG2		LG3		LG4
a		**b**		**c**		**d**
…						
$\{A_1\}$	X	$\{B_1\}$	X	$\{C_2\}$	X	$\{D_2, D_3\}$
$\{A_1\}$	X	$\{B_2\}$	X	$\{C_2\}$	X	$\{D_2, D_3\}$
$\{A_1\}$	X	$\{B_2\}$	X	$\{C_3\}$	X	$\{D_3, D_4\}$
$\{A_2\}$	X	$\{B_2\}$	X	$\{C_2\}$	X	$\{D_3, D_4\}$
$\{A_2\}$	X	$\{B_2\}$	X	$\{C_3\}$	X	$\{D_3, D_4\}$
…						

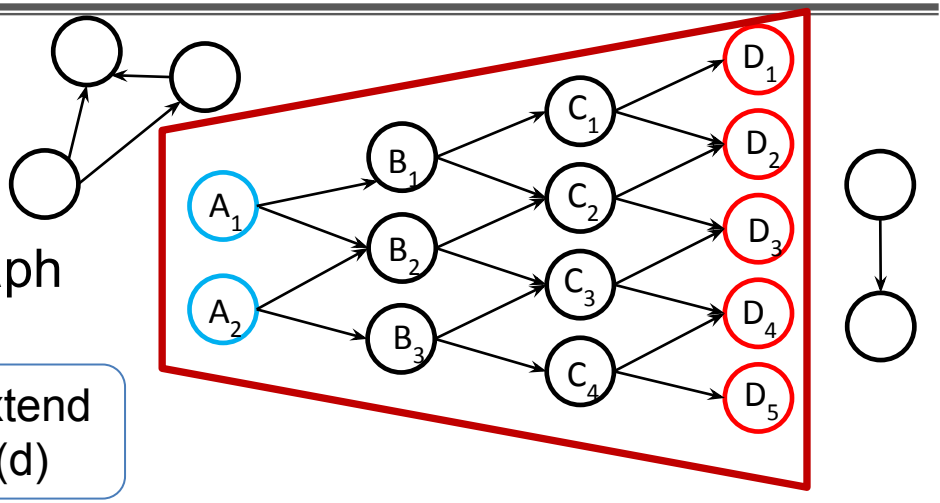
F-Representation

Further Value Repetitions

MATCH



WHERE a.cntr='US' & e.cntr='CAD'



- Most compact output: relevant subgraph

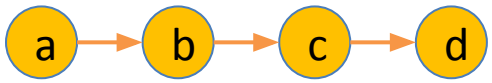
Scan (a) cntr=`US' → Extend (b) → Extend (c) → Extend (d)

LG1		LG2		LG3		LG4
a		**b**		**c**		**d**
…						
$\{A_1\}$	X	$\{B_1\}$	X	$\{C_2\}$	X	$\{D_2, D_3\}$
$\{A_1\}$	X	$\{B_2\}$	X	$\{C_2\}$	X	$\{D_2, D_3\}$
$\{A_1\}$	X	$\{B_2\}$	X	$\{C_3\}$	X	$\{D_3, D_4\}$
$\{A_2\}$	X	$\{B_2\}$	X	$\{C_2\}$	X	$\{D_3, D_4\}$
$\{A_2\}$	X	$\{B_2\}$	X	$\{C_3\}$	X	$\{D_3, D_4\}$
…						

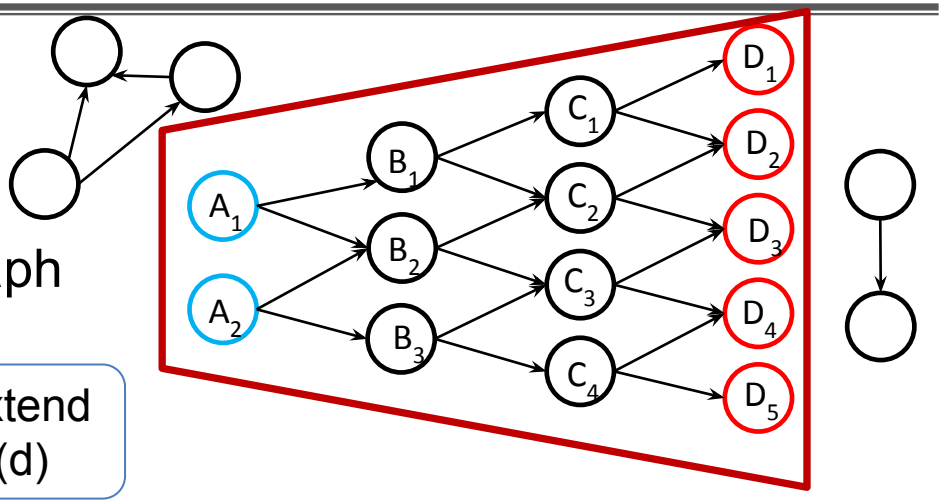
F-Representation

Further Value Repetitions

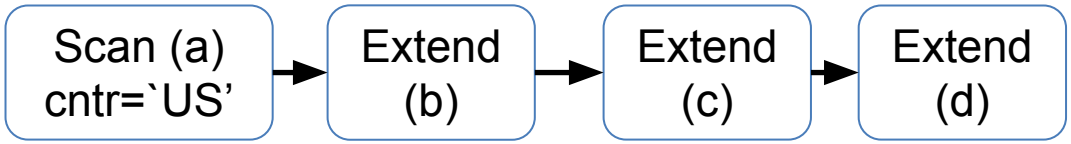
MATCH



WHERE a.cntr='US' & e.cntr='CAD'



- Most compact output: relevant subgraph

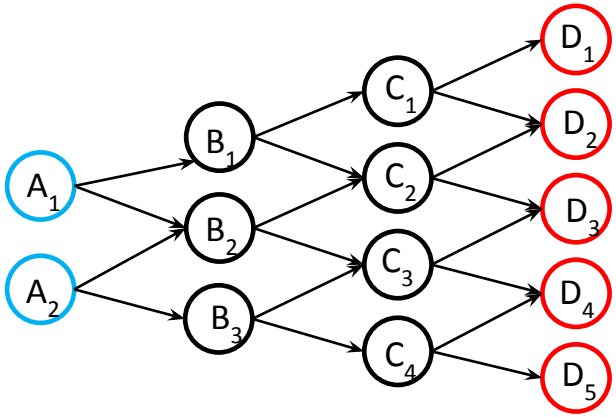


LG1		LG2		LG3		LG4
a		**b**		**c**		**d**
…						
$\{A_1\}$	X	$\{B_1\}$	X	$\{C_2\}$	X	$\{D_2, D_3\}$
$\{A_1\}$	X	$\{B_2\}$	X	$\{C_2\}$	X	$\{D_2, D_3\}$
$\{A_1\}$	X	$\{B_2\}$	X	$\{C_3\}$	X	$\{D_3, D_4\}$
$\{A_2\}$	X	$\{B_2\}$	X	$\{C_2\}$	X	$\{D_3, D_4\}$
$\{A_2\}$	X	$\{B_2\}$	X	$\{C_3\}$	X	$\{D_3, D_4\}$
…						

F-Representation

- F-representations do not capture such repetitions
- Need to group tuples into sets and point to them

D-Representations [Olteanu et al., TODS 2015]



F-Representation

a		b		c		d
…						
$\{A_1\}$	X	$\{B_1\}$	X	$\{C_2\}$	X	$\{D_2, D_3\}$
$\{A_1\}$	X	$\{B_2\}$	X	$\{C_2\}$	X	$\{D_2, D_3\}$
$\{A_1\}$	X	$\{B_2\}$	X	$\{C_3\}$	X	$\{D_3, D_4\}$
$\{A_2\}$	X	$\{B_2\}$	X	$\{C_2\}$	X	$\{D_3, D_4\}$
$\{A_2\}$	X	$\{B_2\}$	X	$\{C_3\}$	X	$\{D_3, D_4\}$
…						

D-Representation

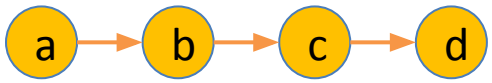
Expr	c		d
DN_{C1}	$\{C_1\}$	X	$\{D_1, D_2\}$
DN_{C2}	$\{C_2\}$	X	$\{D_2, D_3\}$
DN_{C3}	$\{C_3\}$	X	$\{D_3, D_4\}$
DN_{C4}	$\{C_4\}$	X	$\{D_4, D_5\}$

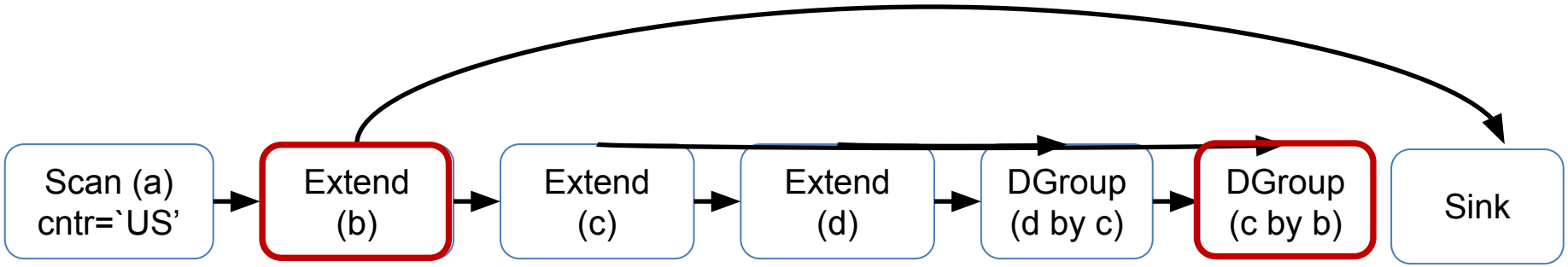
Expr	b		{c, d}
DN_{B1}	$\{B_1\}$	X	$\{DN_{C1}, DN_{C2}\}$
DN_{B2}	$\{B_2\}$	X	$\{DN_{C2}, DN_{C3}\}$
DN_{B3}	$\{B_3\}$	X	$\{DN_{C3}, DN_{C4}\}$

a		{b, c, d}
$\{A_1\}$	X	$\{DN_{B1}, DN_{B2}\}$
$\{A_2\}$	X	$\{DN_{B2}, DN_{B3}\}$

D-Representation-based Processor (1)

```
MATCH
WHERE a.cntr='US' & e.cntr='CAD'
```



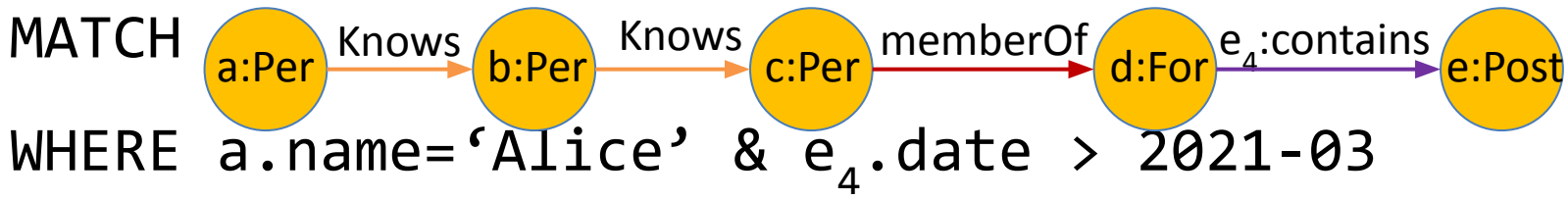


Expr	b		{c, d}
DN_{B1}	$\{B_1\}$	X	$\{DN_{C1}, DN_{C2}\}$
DN_{B2}	$\{B_2\}$	X	$\{DN_{C2}, DN_{C3}\}$
DN_{B3}	$\{B_3\}$	X	$\{DN_{C3}, DN_{C4}\}$

- DGroup: Constructs and puts D-reps into shared caches with join operators
- DAG style plans: operator to skip sub-plans when there is a cache hit
- Every other operator is list-based, so operates on blocks

D-Representation-based Processor (2)

- LDBC Social Network Benchmark, Scale factor 10, 176M edges
- Ex:



- GF-FRep vs GF-DRep

	IC1	IC2	IC3	IC4	IC5	IC6	IC7	IC8	IC9	IC10
GF-FRep	7.1	6.5	80.4	0.36	188.0	21.0	0.17	0.05	181.5	0.85
GF-DRep	0.2 **30.0x**	3.0 **2.2x**	5.2 **15.3x**	0.40 **0.9x**	5.8 **32.1x**	0.9 **23.0x**	0.17 **1.0x**	0.07 **0.7x**	9.7 **18.7x**	0.21 **4.1x**

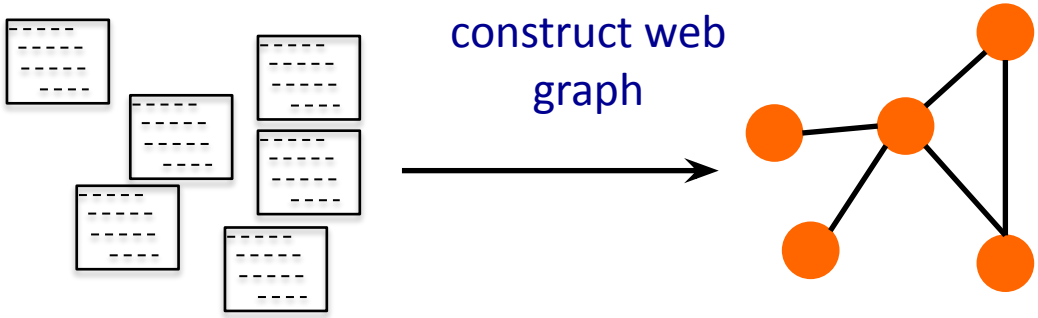
runtimes in seconds

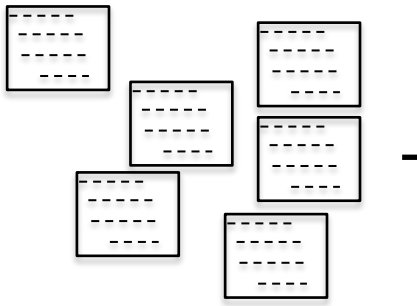
Outline

- Courses/Topics Beyond 348
- GraphflowDB: Graph Database Management Systems
- MapReduce

Google's Large-scale Applications

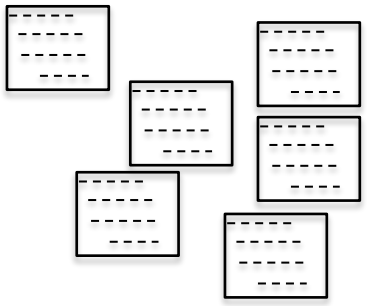
- Many applications processing very large data: (in Petabytes)
- Ex: Trillions of web pages, ad clicks, queries, videos, images





Host	# Pages
cnn.com	1300
yahoo.com	4845
uwaterloo.ca	762
techcrunch.com	2582
…	…

Google's Large-scale Applications

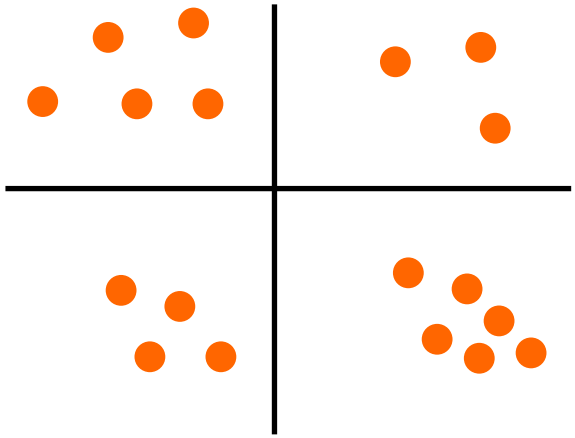


inverted index

word	Pages
trudeau	cbc.ca, nytimes.com, wikipedia.org, …
apple	apple.com, farming.ca, …
computer	ibm.com, wikipedia.org,…
…	…

user	clicks
user1	cbc.ca
user1	techcrunch.com
user2	wikipedia.com
user3	uwaterloo.ca
user3	abc.ca
…	…

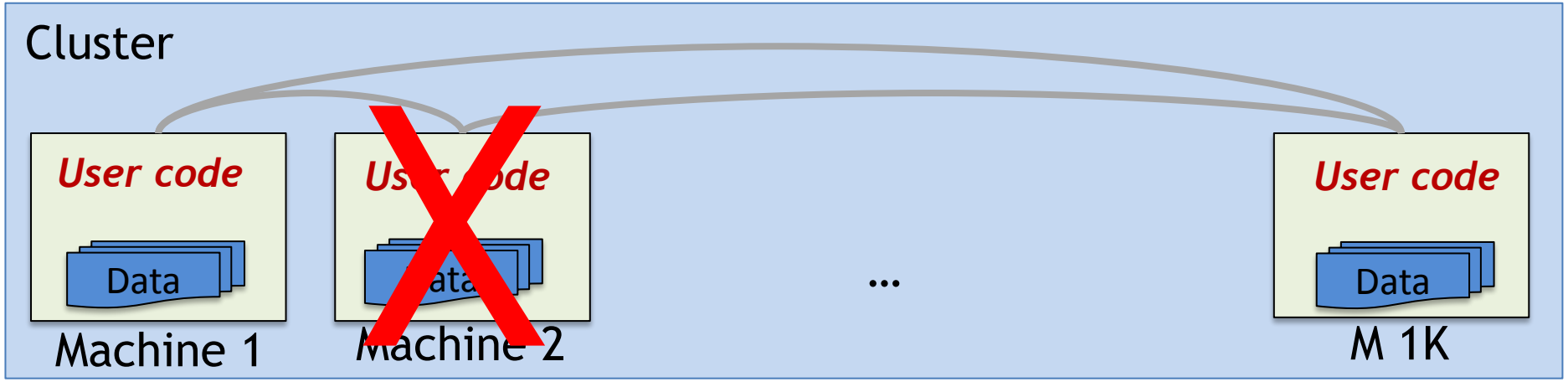
clustering "similar" users (recommendation)



- Many others: Ad clicks, videos, images, internal machine logs, etc…

Google's Problem

- Need 100s/1000s machines to process such large data.



- But: Writing distributed code is very difficult! Need code for:
  - Partitioning data.
  - Parallelizing computation.
  - Coordinating the progress.
  - Communication of machines.
  - Dealing with *machine failures*!
  - Others: load-balancing, etc.
- All much more complex than the simple application logic!
  - E.g: Take each web doc. & spit out an edge for each hyperlink

“Law of Large-Scale Distributed Systems”

- Law of Hardware: Like humans, hardwares live & die
- Consider a super reliable machine: Mean-time-btw-failure: 30 years
- At any day prob. failure: ~1/10000
- Consider a distributed system running on 10000 machines

Q: Prob a machine will die at any day?

A: 64%

Fact of Life: Machines will fail frequently at large-scale.*

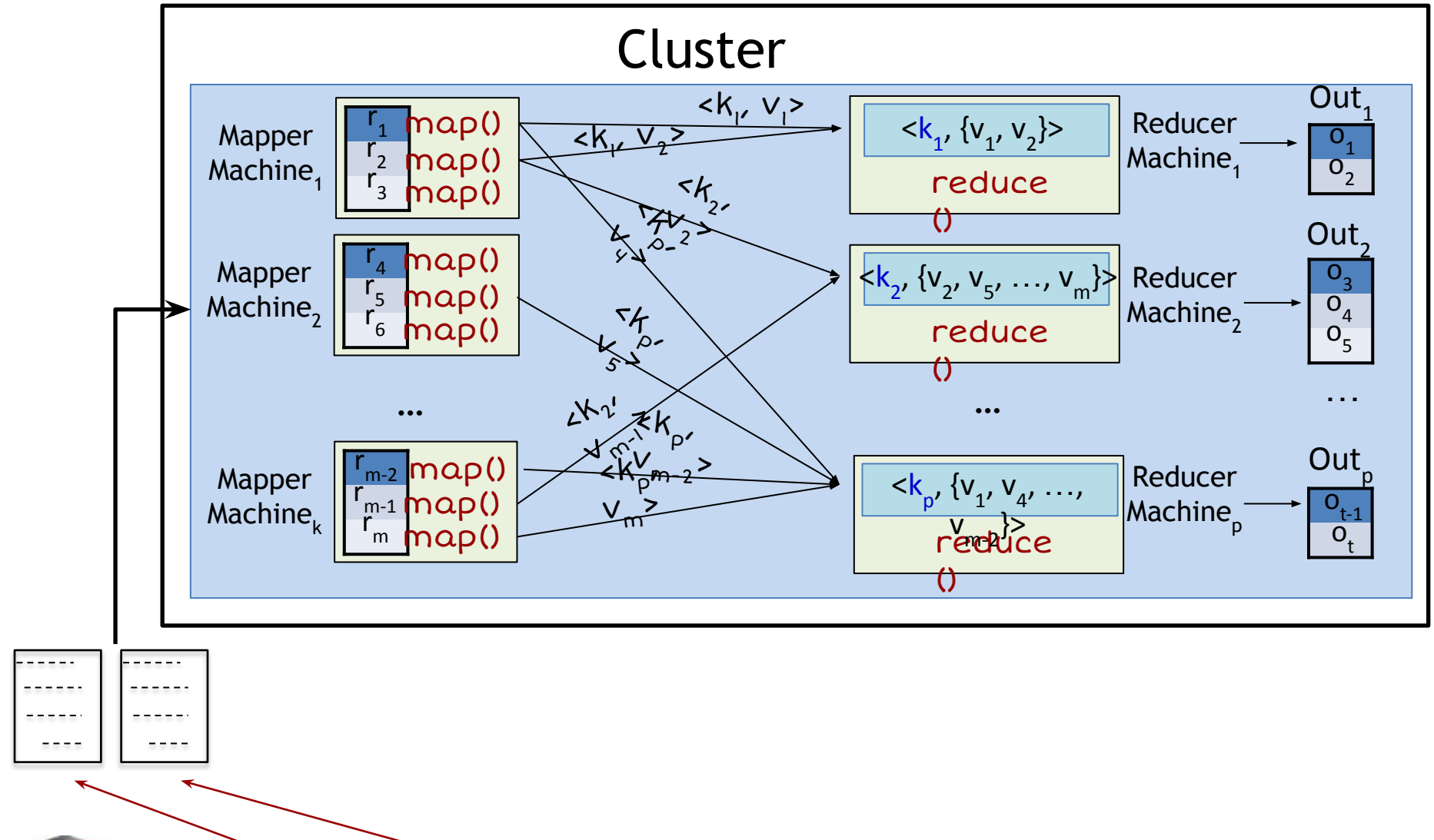
Google's Previous Solution

- Special-purpose distributed program for each application.
- Problem: Not scalable
  - Difficult for many engineers to write parallel code
  - Development speed: Thousands lines for simple applications.
  - Code repetition for common tasks:
    - Data Partitioning
    - Code parallelizing, Fault-tolerance etc.

Google's New Solution: MapReduce

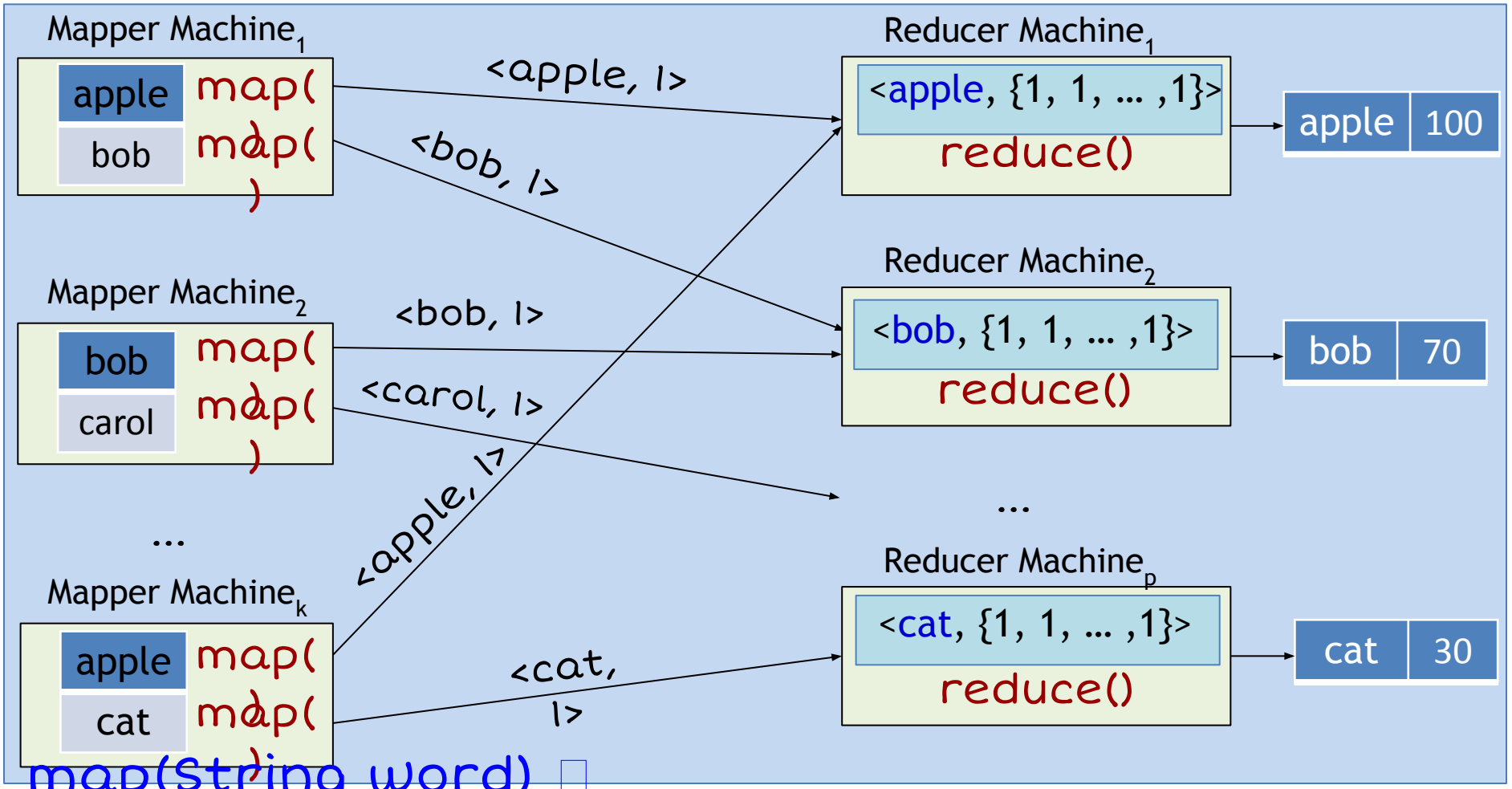
- Distr. Data Processing System with 3 fundamental properties:
  1. Transparent Parallelism
     - i.e: User-code parallelization & data partitioning automatically done by the system
  2. Transparent Fault-tolerance
     - i.e: Recovery from failures is automatic
  3. A Simple Programming API
     - map() & reduce()

MapReduce Overview



map() reduce(
functio)
p functio

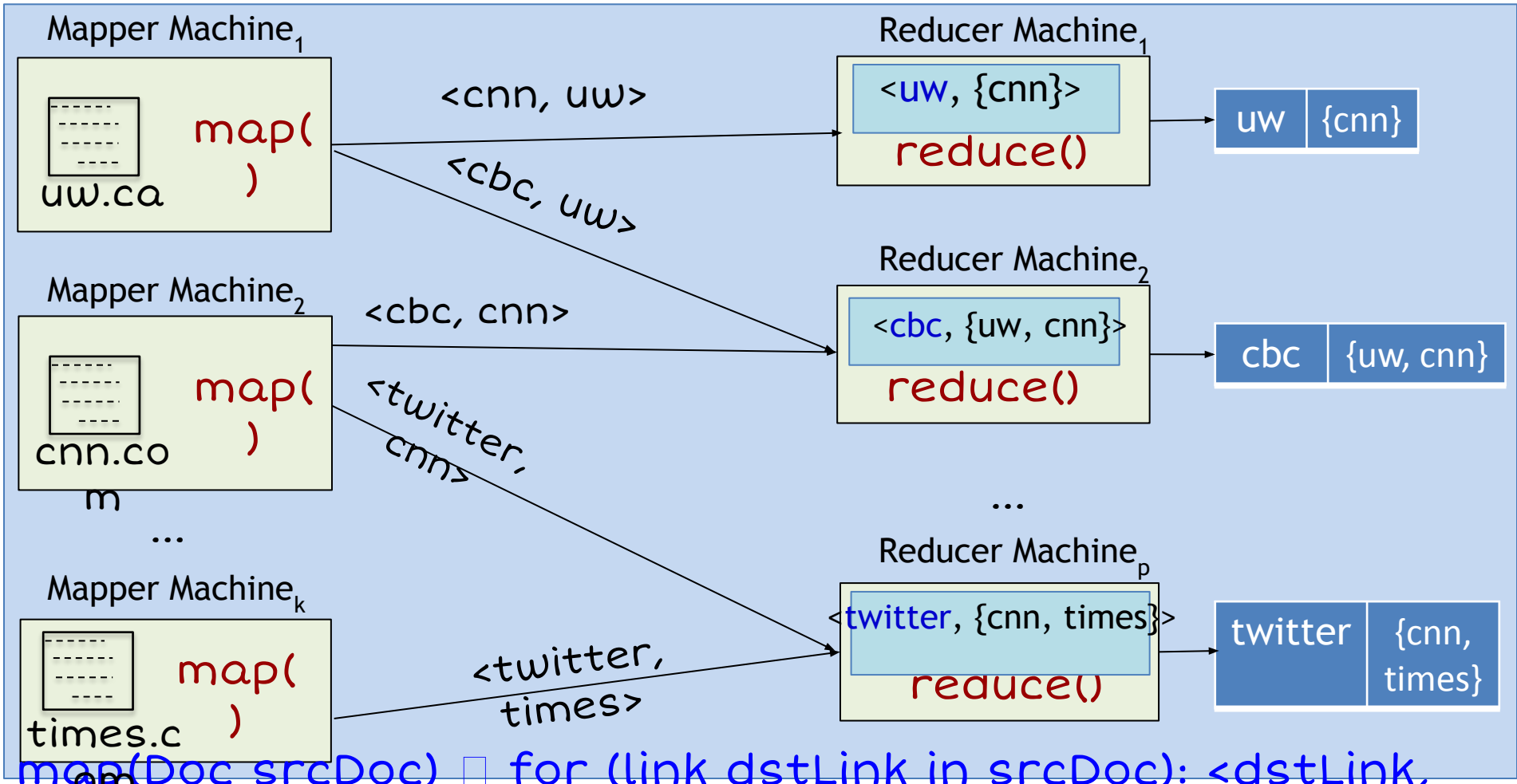
Example 1: Word Count



map(String word) □

reduce(String word, List<Integer> ones): □ <word, sum(ones)>

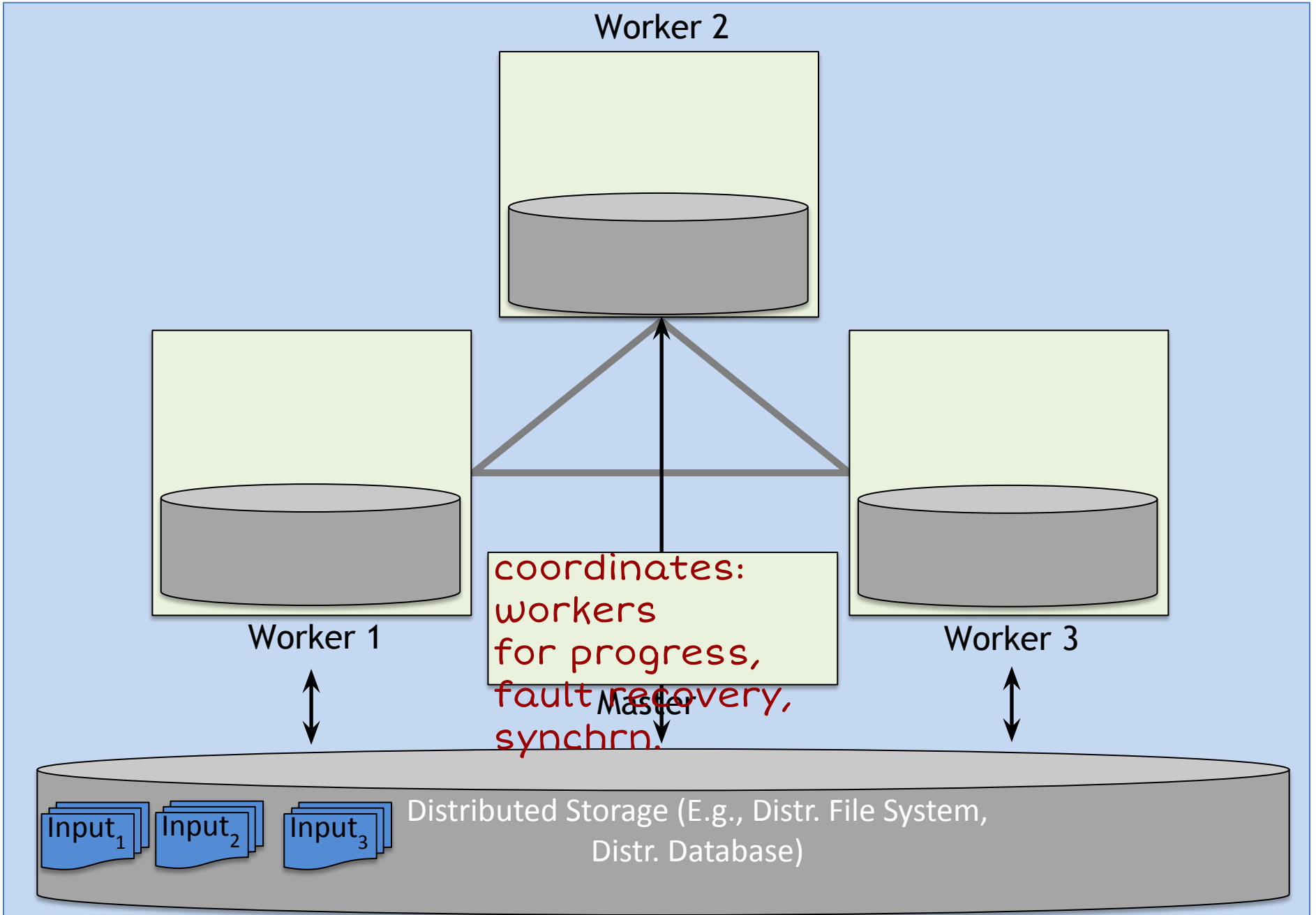
Example 2: Reversing Web-Link Graph



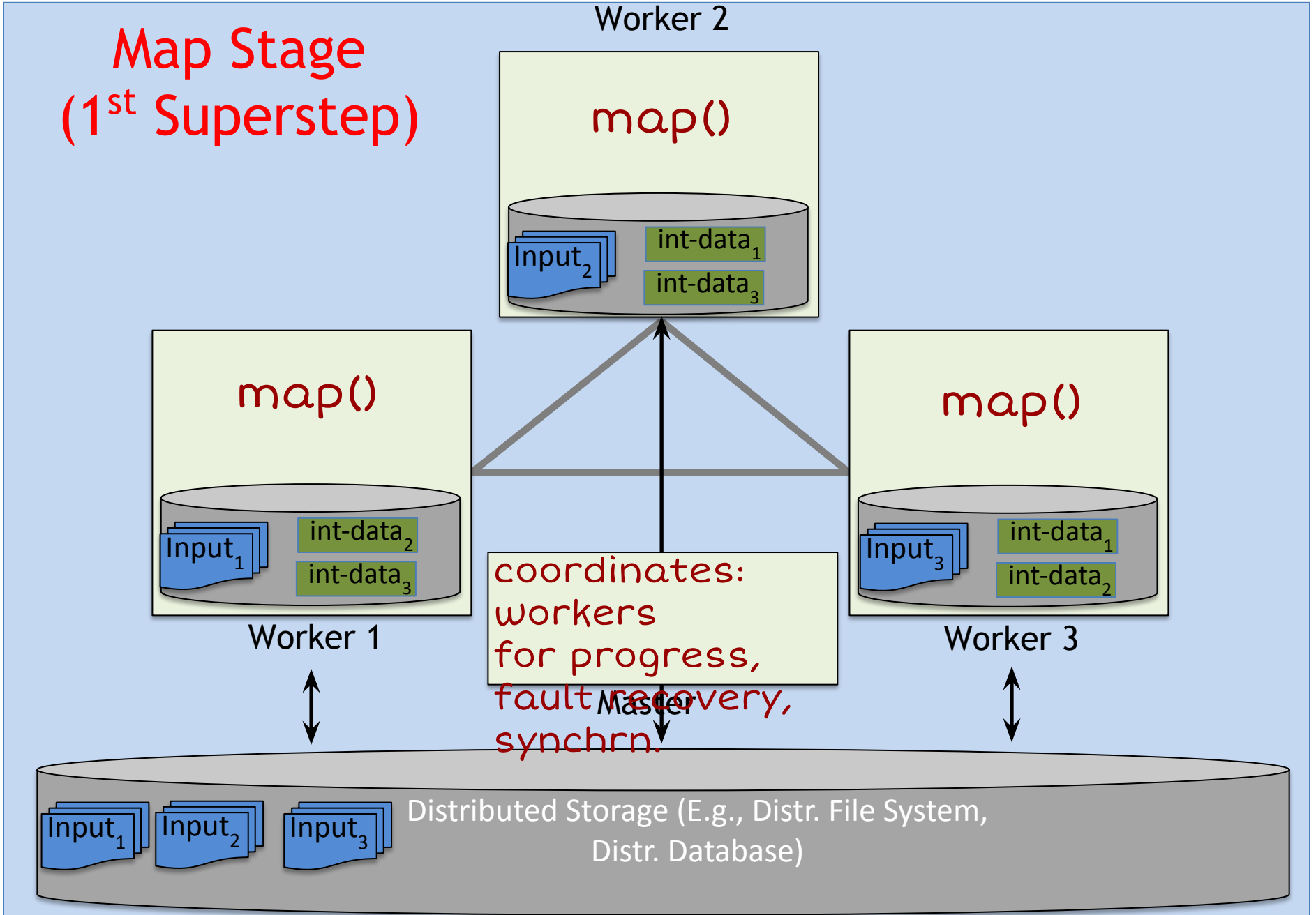
map(Doc srcDoc) → for (link dstLink in srcDoc): <dstLink, srcDoc.link>

reduce(String target, List<Long> srcs): → <target, srcs>

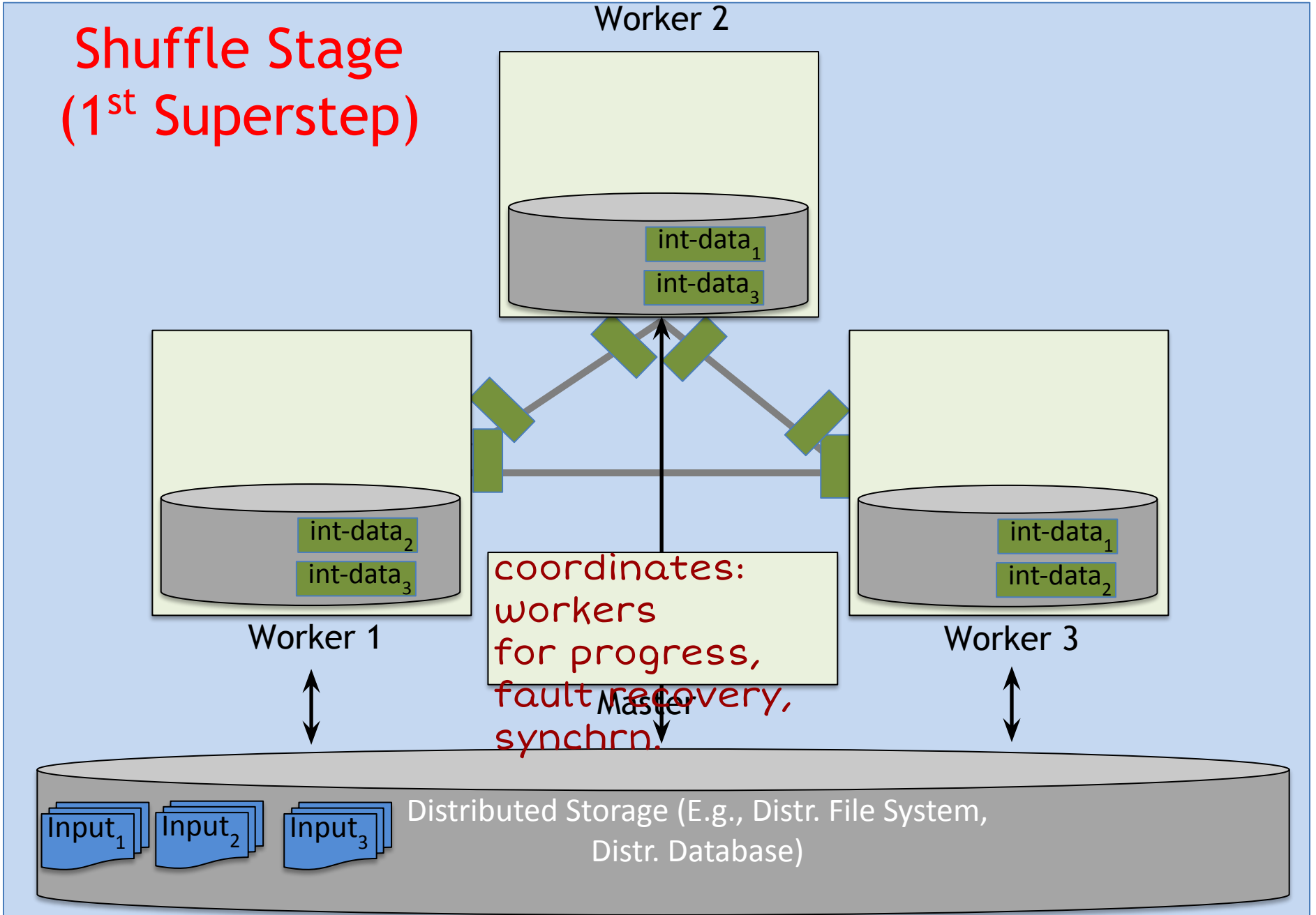
More Detailed Architecture



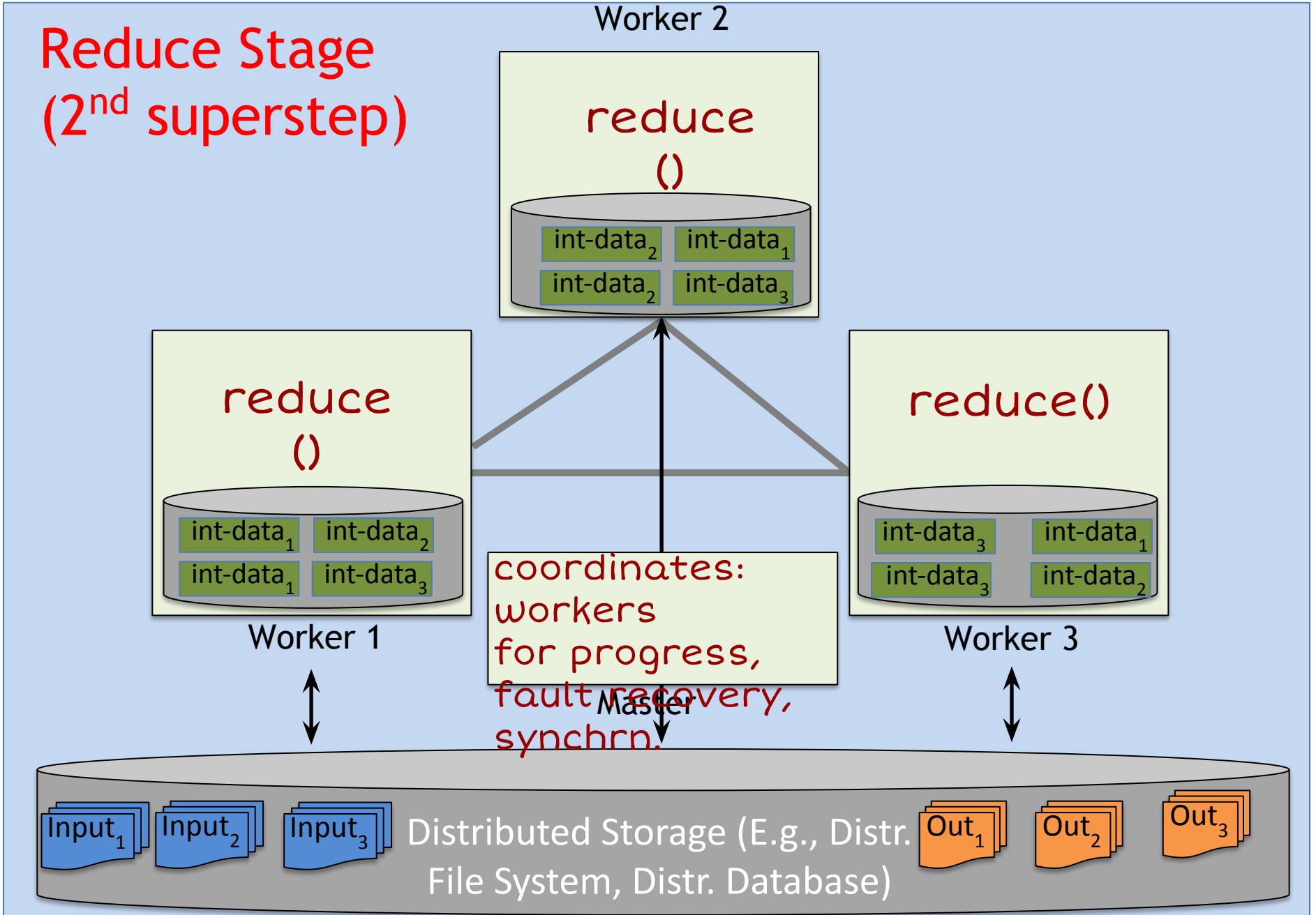
More Detailed Architecture



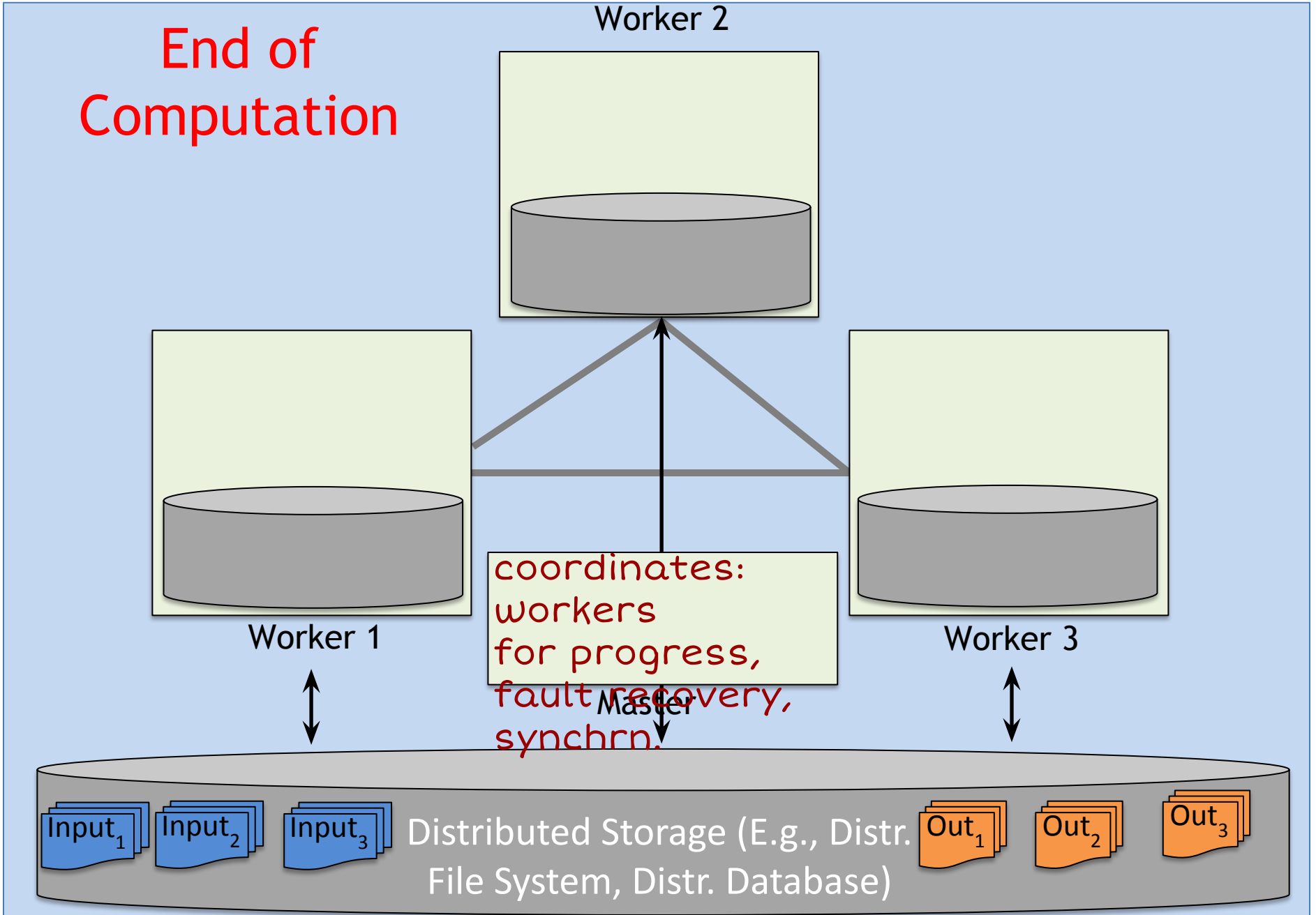
More Detailed Architecture



More Detailed Architecture



More Detailed Architecture



Acknowledgements

- Prof Xi He!
- All TAs: Guy Coccimiglio, Amine Mhedhbi, Karl Knopf, Chang Liu, Glaucia Melo, Jessica Saini
- Sylvie Davies

Thank you!