CMSC 724: Database Management Systems
Models, Languages, and Abstractions

Instructor: Amol Deshpande
amol@cs.umd.edu

Outline

- Data Models: Then, and now
  - History of Data Models ("what comes around…")
- Languages
  - Overview
  - Datalog ("a survey of research…" and "declarative networking…")
- Map-reduce and Spark
  - Original MR Abstraction ("mapreduce:" …")
  - Spark ("resilient distributed datasets…")
- SystemML: An abstraction for ML
- GraphX: An abstraction for Graphs

Things to Think About

- Goal is to choose a good data model for the data
  - Needs to be sufficient expressive – should capture real-world data
  - Easy to use for users – support physical and logical data independence
  - Lends to good performance
- Many similarities across models
  - Much convergence in the last two decades
- Keep in mind orthogonal issues of schema maintenance and evolution, and data integration/reconciliation
  - Possibly a much bigger headache in practice
- No doubt relational is the best low-level model, but should it continue to be the high-level model as well?

Physical & Logical Data Independence

- Examples of Physical Data Dependence (from Ted Codd, 1970)
  - Ordering dependence: How records are sorted hard-coded into the apps
  - Indexing dependence: What indexes are present on the data hard-coded in the apps
  - Access Path dependence: Dependence on the hierarchy or the network model chosen

- Logical data independence
  - Ability to make changes to the schema, e.g., add a new attribute, combine two tables, etc., without affecting external applications or APIs
  - Can be achieved through use of “views” in RDBMSs

- In general: we want the application programs to not hard-code any of those decisions so those can be changed easily

Running Example

Supplier (sno, sname, scity, sstate)
Part (pno, pname, psize, pcolor)
Supply (sno, pno, qty, price)

A Relational Schema
Figure 1

Suppplier

16 General Supply Boston Ma
24 Special Supply Detroit Mi

Part

27 Power saw 7 silver
42 bolts 12 gray

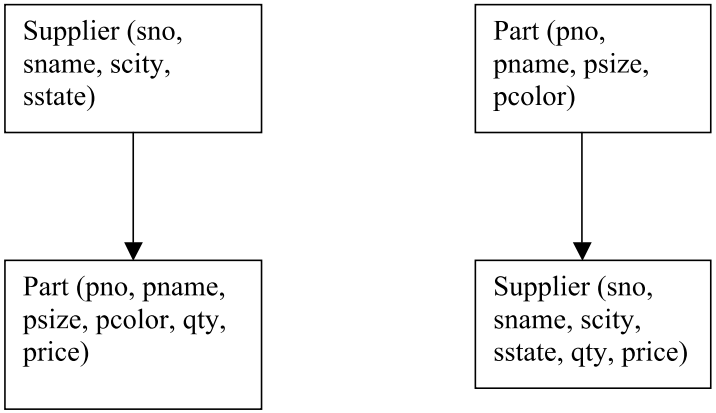
Supply

16 27 100 $20.00
16 42 1000 $.10
24 42 5000 $.08

Some Sample Data
Figure 2

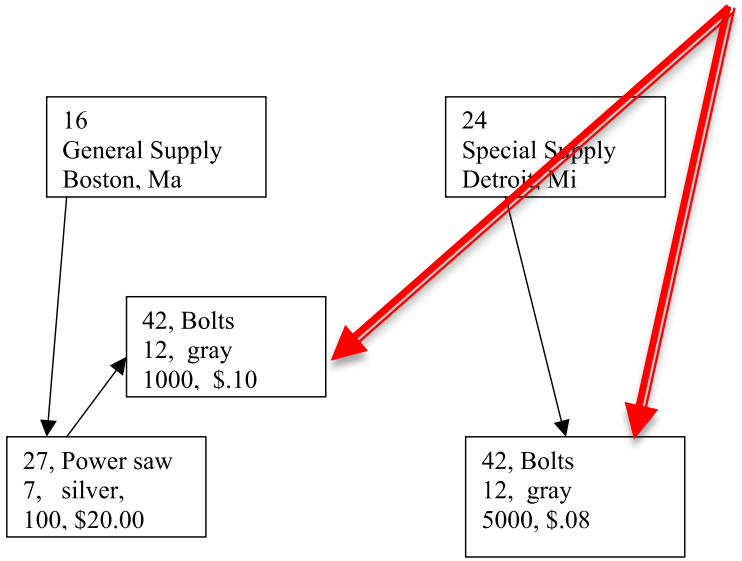
IMS Era (60's)

- Main constructs: "record type" (schema), "instance" (must obey schema), "keys" (to uniquely identify records)
- Record types must be arranged in a hierarchy
  - Record instances stored using the same hierarchy
  - Records with the same parent stored as a linked list



Two Hierarchical Organizations
Figure 3

Repeated information



Some Example Data
Figure 4
for the first hierarchy

Can't store a part not supplied by anybody

IMS Era (60's)

- "Record-at-a-time" query language
  - Programmer had to keep track of the "currency" indicators in their program – leads to complex edge cases

```
Get unique Supplier (sno = 16)
Until no-more {
        Get next within parent (color = red)
}
```

```
Until no-more {
        Get next Part (color = red)
}
```

- Multiple storage options (e.g., sequential, using a B+-Tree, using Hashing)
- Supported some "physical data independence"
  - DL/1 language was written against the logical hierarchy to some extent
  - But use of hashing vs indexing still important (e.g., can't use "get-next" on hashing storage)
- Several hacks added later to support non-tree structured data
  - Adds much more complexity

IMS Era (60's)

- Some Lessons
  - Physical and logical data independence are highly desired
  - Tree-structured data models restrictive – not general enough, and hard to modify
  - Manual query optimization unlikely to work over long term

- We see the same issues with JSON and XML databases of today
  - No logical data independence – the hierarchies get hard-coded into queries
  - Significant physical data independence today though

```
{"id": "16",
 "Name": "General Supply",
"Location": "Boston, MA",
"supplies": [
        {"id": "27", "Name": "Power Saw", "Qty": 7, "Color": "gray"},
        {……}
]},
{"id": "24",
 "Name": "Special Supply",
"Location": "Detroit, MI",
"supplies": [
        {"id": "27", "Name": "Power Saw", "Qty": 10, "Color": "gray"},
        {……}
]}
```

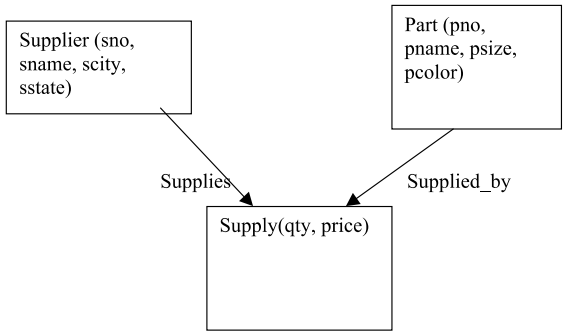
VS

```
Collection "Suppliers":
{"id": "16",
 "Name": "General Supply",
"Location": "Boston, MA",
"supplies": [
        {"id": "27", "Qty": 7},
        {……}
]},

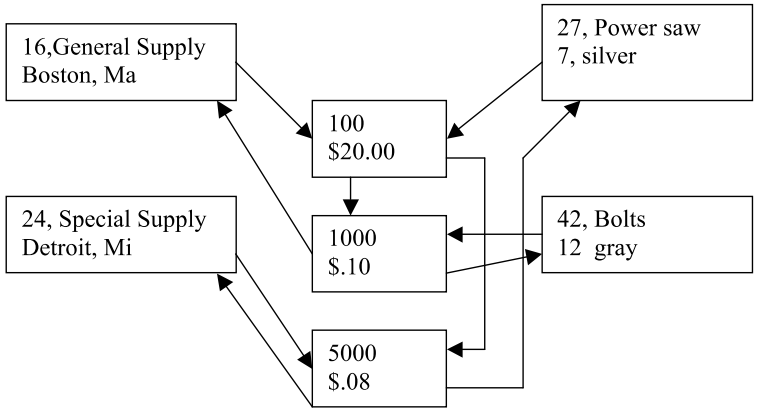
Collection "Parts":
{"id": "27",
  "Name": "Power Saw",
  "Color": "silver"
}
```

CODASYL Era (70's)

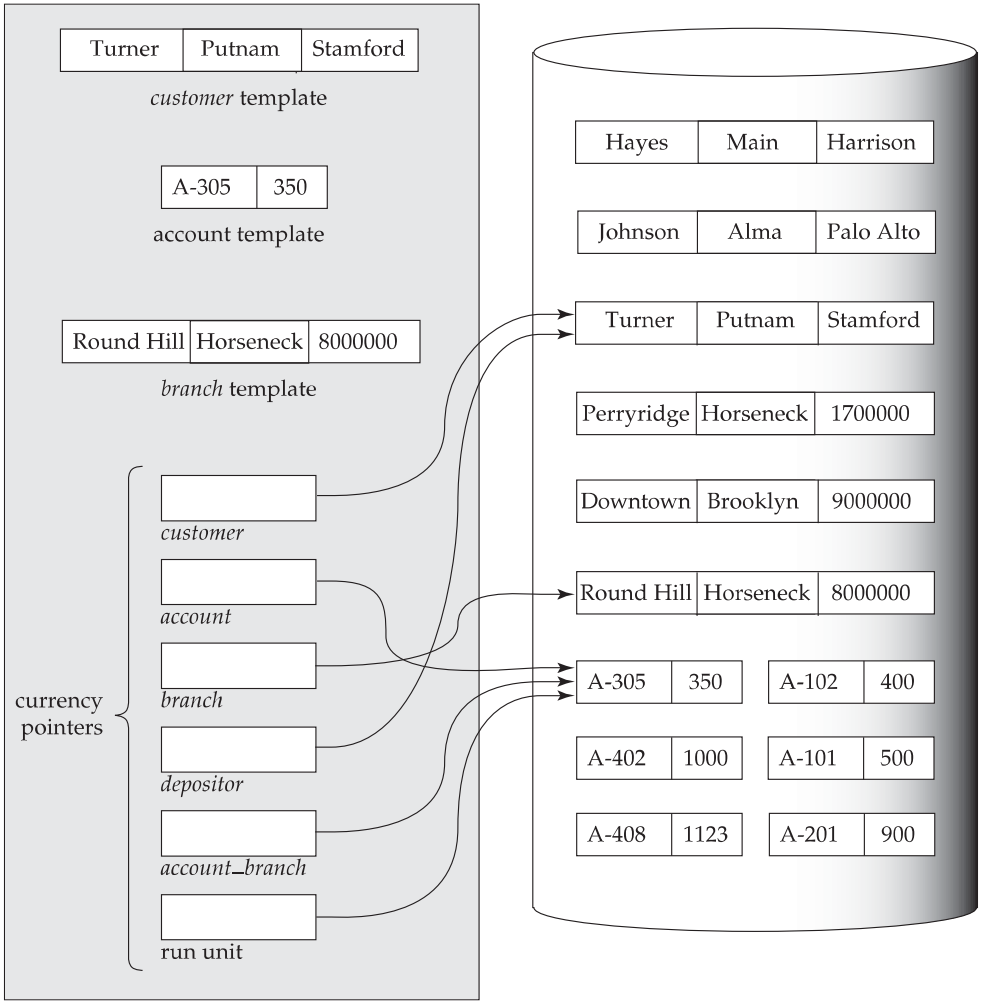
- Directed Graph Data Model, with a "record-at-a-time" data manipulation language
- Fewer restrictions than the IMS model
- But much more complex for the programmer ("programmer as a navigator")
- Limited physical or logical data independence
- Harder to do bulk-loading of data



A CODASYL Directed Graph
Figure 7



Some Example Data
Figure 8




```
customer.customer_city := "Harrison";
find any customer using customer_city;
while DB-status = 0 do
    begin
        get customer;
        print (customer.customer_name);
        find duplicate customer using customer_c
    end;
```

Figure A.20 Program work area.

CODASYL Era (70's)

- Very similar to today's graph data model proposals (e.g., "property graph")
- But those show significantly more physical and logical data independence
  - Depending on the actual implementation
  - Need to enforce schemas (many graph databases today don't)
- Many of the identified limitations of CODASYL really about the language and some implementation choices

- Also bears much similarity with Entity-Relational Model (at the conceptual level)
  - E/R Model never really had an implementation or a language

Relational Era

- Proposed by Ted Codd in 1969/1970
  - "IMS programmers were spending large amount of time doing maintenance on IMS applications when logical or physical changes occurred"
- Proposal:
  - Store data in a simple data structure (tables)
  - Access it through a high-level set-at-a-time DML (relational algebra → SQL)
  - No need to mandate any physical storage design (each system can do its own, and change easily as needed)
- Can easily represent 3-entity relationships (difficult for network model)
- No existence dependencies that plagued hierarchical model
- Cons:
  - Transitive closure
  - (initially) performance
  - (initially) too complex and mathematical languages

Relational Era

- Many debates in 1970's
- Relational Model Advocates
  - Nothing as complex as CODASYL can possibly be a good idea
  - CODASYL does not provide acceptable data independence
  - Record-at-a-time programming is too hard to optimize
  - CODASYL and IMS are not flexible enough to easily represent common situations (such as marriage ceremonies)
- CODASYL Advocates
  - COBOL programmers cannot possibly understand the new-fangled relational languages
  - It is impossible to implement the relational model efficiently
  - CODASYL can represent tables, so what's the big deal?
- Both camps changed positions to move towards each other
  - Relational systems got user-friendly languages (SQL, QUEL), and efficient implementation
- (According to Authors) Effectively settled by mini-computer revolution, and by IBM who announced new relational products
  - And by non-portability of CODASYL engines

Relational Era

- Don Chamberlin of IBM was an early CODASYL advocate (later co-invented SQL)

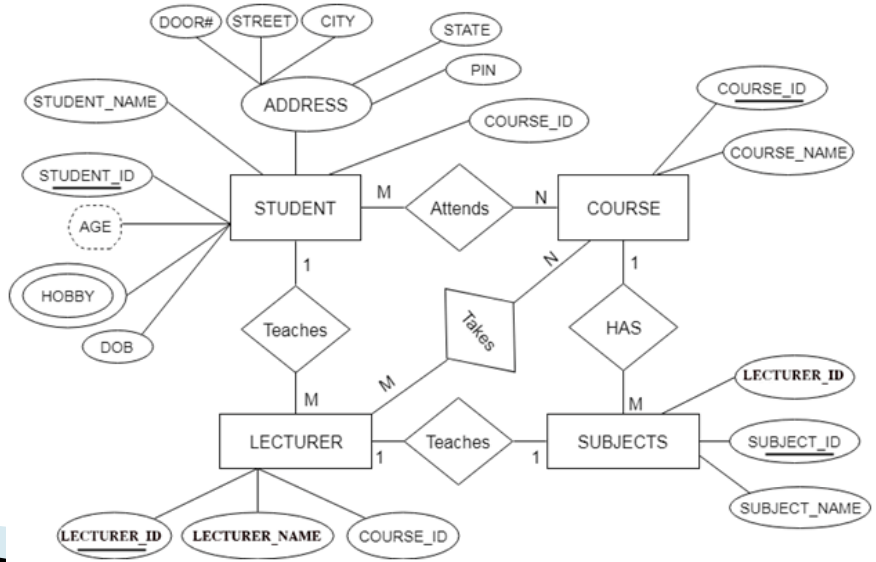
“He (Codd) gave a seminar and a lot of us went to listen to him. This was as I say a revelation for me because Codd had a bunch of queries that were fairly complicated queries and since I’d been studying CODASYL, I could imagine how those queries would have been represented in CODASYL by programs that were five pages long that would navigate through this labyrinth of pointers and stuff. Codd would sort of write them down as one-liners. These would be queries like, "Find the employees who earn more than their managers." [laughter] He just whacked them out and you could sort of read them, and they weren’t complicated at all, and I said, "Wow." This was kind of a conversion experience for me, that I understood what the relational thing was about after that.”

Entity-Relational Model

- Explicitly represent entities and relationships, and connections between them
  - Much easier for conceptual development of a schema
- No real uptake as the physical data model used by a database back then
  - Lot of similarities to CODASYL
  - Easy to map to relational
- Widely used today for initial schema design
  - Normal forms are too difficult to work with
  - Don't address the question of how to get started



An E-R Diagram
Figure 11



Entity-Relational Model

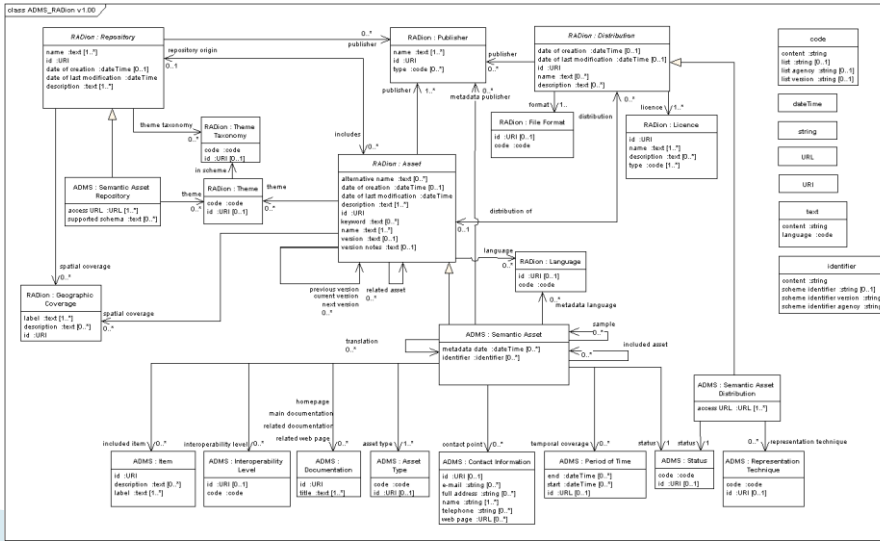
- Many similarities to Object-relational Mapping Frameworks (like ruby-on-rails, python Django, etc).
  - Those frameworks explicitly model ”object types” and “relationships” between them
  - Very widely used by application programmers today
  - Typically mapped to a RDBMS at the backend (not always a faithful mapping)
  - Also similar to “property graphs” (assuming schemas are enforced)
- In my opinion: E/R model should be resurrected as the primary model for RDBMS
  - Maintenance of relational model is too hard
  - Changes made over time lead to un-normalized schemas with many issues
  - See “Database Decay” by Stonebraker et al.

```python
class Question(models.Model):
    question = models.CharField(max_length = 128, blank = False)
    asker = models.ForeignKey(CustomUser, on_delete = models.CASCADE, related_name="asked_questions")
    requested = models.ManyToManyField(CustomUser, related_name="answer_requests")
    genres = models.ManyToManyField(Genre, related_name = "questions")
    followers = models.ManyToManyField(CustomUser, related_name="followed_questions")
    topics = models.ManyToManyField(Topic, related_name="questions")
    url = models.CharField(max_length = 150)
    created_at = models.DateTimeField(auto_now_add = True, blank = True)

    class Meta:
        ordering = ('-created_at', )

    def __str__(self):
        return self.question

class Answer(models.Model):
    answer = models.TextField()
    author = models.ForeignKey(CustomUser, on_delete = models.CASCADE, related_name="written_answers")
    question = models.ForeignKey(Question, on_delete = models.CASCADE, related_name="answers")
    upvoters = models.ManyToManyField(CustomUser, related_name="upvoted_answers")
    created_at = models.DateTimeField(auto_now_add = True, blank = True)
```



R++ Era

- Many new proposals focusing on specific applications that were not a good fit for relational
  - CAD, Text Management, Time, Graphics, etc.

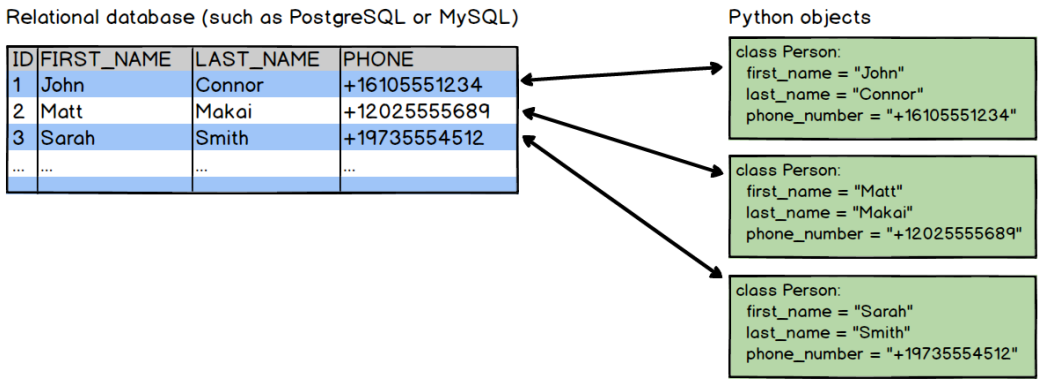
- GEM [Zaniolo 83]
  - Set-valued attributes (e.g., available colors in “parts”)
  - Cascaded dot notation (e.g., how you do in ORMs)
  - Inheritance hierarchies

```
Select Supply.SR.sno
From Supply
Where Supply.PT.pcolor = “red”
```

- Main cons:
  - No real improvements over the relational model, either functionality or performance
  - Some of the key constructs could be easily added to relational model (e.g., new data types, arrays)

Object-oriented Models

- Designed to handle the “impedance mismatch”
  - How data is represented in memory (typically as objects) vs how it is stored (in a normalized relational schema)
- Essentially became “persistent” programming languages
  - Interesting technical challenge: “pointer swizzling”
- Weak support for transactions, queries, etc.
  - Largely single-user systems
  - DBMS must run in the same address space as the application
- Several reasons didn’t succeed
  - No major additional functionality for most applications (i.e., a niche market)
  - No standards
  - Too tied to a single programming language
- Bear many similarities to Graph Databases
  - OrientDB, one of the major graph databases, basically an Object Store



Object-relational Models

- Motivated by need to represent more complex data types
  - e.g., locations, rectangles, complex numbers etc.
- Possible to do in relational to some extent, but very painful and error-prone
- Instead, have:
  - User-defined types
  - User-defined operators (that change the meaning of "+")
  - User-defined functions to work on those types (e.g., to find if a Point lies in a Rectangle)
  - User-defined "indexes" (for efficient searching): B+-trees don't work well on spatial data

```
Select P-id
From Parcel
Where  Xmax > X0 and  Ymax > Y0 and Xmin <  X1 and Ymax < Y1
```

VS

```
Select P-id
From Parcel
Where  P-box ## "X0, X1, Y0, Y1"
```

- Postgres (research project at Berkeley, led by Stonebraker) the first real implementation
- UDFs a major benefit
  - By putting code in the databases, avoided many round-trips to the database from the client

Semi-structured Data

- Context: XML very popular in the industry and academia circa 2000
  - When this article was first written
- Flexible schemas ("schema later" or "schema-on-read")
  - Don't require a schema in advance – instead impose it when reading, or make it part of the data itself (self-describing)
  - Relational databases will reject any data that doesn't conform to the schema
  - Easy to state, but hard to use due to semantic heterogeneity and other issues
  - Need a well-defined understanding of what the fields mean to use it properly (i.e., need a schema)
- These databases also support easy schema evolution, which is a major benefit especially in early application development

```
Person:
        Name: Joe Jones
        Wages: 14.75
        Employer: My_accounting
        Hobbies: skiing, bicycling
        Works for:  ref (Fred Smith)
        Favorite joke: Why did the chicken cross the road?
        Office number: 247
        Major skill:  accountant
End Person
```

```
Person:
        Name: Smith, Vanessa
        Wages: 2000
        Favorite coffee:  Arabian
        Pastimes: sewing, swimming
        Works_for:  Between jobs
        Favorite restaurant:  Panera
        Number of children:  3
End Person:
```

Semi-structured Data

- Context: XML very popular in the industry and academia circa 2000
  - When this article was first written
- Main constructs in XML (and JSON today, mostly)
  - ”Self-describing” (the attribute names part of the data)
  - Hierarchical format
  - Can have links to other records (like CODASYL)
  - Set-valued attributes (leads to more complex language)
- XML popular early on as a “on-the-wire format” (as is JSON today)
  - Text-based, so can go through firewalls
  - Not proprietary
  - JSON widely used for APIs today
  - Other formats like Parquet are more common for large volumes of data

Developments since 2005

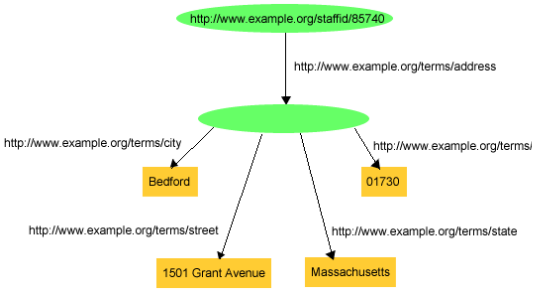
- From redbook.io: http://www.redbook.io/ch1-background.html

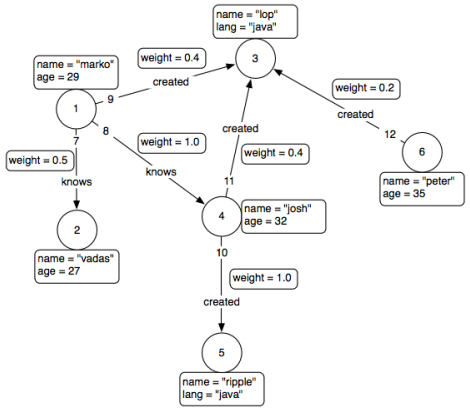
- XML didn't really take over
  - Too complex, especially query languages (XQuery)
  - Inefficient implementations (e.g., for indexes)
  - No compelling use cases

- JSON more popular and very similar
  - Good for sparse data – can be handled by adding a JSON data type to the database
  - Schema on read – basically just a "project" on fewer attributes of interest
  - Doesn't really solve semantic heterogeneity problems

NoSQL Data Models

- Quite a few different data models, but map closely to one of the standard ones
- Document Data Model: JSON, XML, etc.
- Key-Value: Simple interface
  - put(key, value), get(key)
  - Lot of logic in what "value" is and how it is manipulated
  - e.g., value may be JSON, time-series, etc.
- Graph Data Models:
  - A few different ones: Property Graph, RDF, …
- Column families (e.g., Cassandra)
  - Puts a little more structure on key-value
  - "values" themselves are stored as columns of information
- Array Data Models
  - Proposed for scientific data management
  - Key abstraction: an array or a matrix
  - Key operations: Slicing, Subsetting, Filtering, etc.





J \ I	[0]	[1]	[2]	[3]	[4]
[0]	(2, 0.7)	(5, 0.5)	(4, 0.9)	(2, 0.8)	(1, 0.2)
[1]	(5, 0.5)	(3, 0.5)	(5, 0.9)	(5, 0.5)	(5, 0.5)
[2]	(4, 0.3)	(6, 0.1)	(6, 0.5)	(2, 0.1)	(7, 0.4)
[3]	(4, 0.25)	(6, 0.45)	(6, 0.3)	(1, 0.1)	(0, 0.3)
[4]	(6, 0.5)	(1, 0.6)	(5, 0.5)	(2, 0.15)	(2, 0.4)

Figure 1: Simple Two Dimensional SciDB Array.

What goes around… and around…

- Stonebraker & **Pavlo**, SIGMOD Record, June 2024
- Sequel to the 2005 Red Book chapter
- 2005: 40 years, nine models → object-relational won
- 2024: 20 more years, same question
- Two halves: **8 data models · 6 architectures**

What goes around… and around…

- Relational model still dominates
- SQL **absorbed** its challengers again
- "Don't use SQL!" → **"Not only SQL"**
- JSON in SQL:2016
- Arrays + property graphs in SQL:2023

MapReduce and Key/Value

- MR: a **point solution** for one crawl
  - SELECT map() FROM t GROUP BY reduce()
- Google killed it in 2014

- KV: the minimal model — (key, value)
- Both **dissolved into** other systems

Document and Column-Family

- Documents = OO (1980s) + XML (1990s), again
- The pitch: joins are slow, ACID is optional
- Real cause: **OLTP couldn't scale in the 2000s**

- Denormalization critique dates to the **1970s**

- MongoDB added SQL in 2021 — last holdout
- Column-family: documents with **one level** of nesting

Text, Array, Vector

- Text: no data model — extract structure

- Arrays: **the one real loss** to the RM
- SQL/MDA lands in SQL:2023 anyway

- Vectors = 1-D arrays, different *queries*
  - "An index is a feature, **not an architecture**"

- JSON took years to absorb; vectors took **months**

Graph Databases

- RDF triples vs. property graphs
- Neo4j chases pointers — **like CODASYL**
- Any graph simulates as Node + Edge
- SQL/PGQ in SQL:2023
- DuckDB beat a graph DBMS by **up to 10×**

The Implementation Changed; the Model Didn't

- **Columnar** — took over OLAP outright
- **Cloud** — networks outran disks; shared-disk returns
- **Lakehouse** — open formats, DBMS no longer gatekeeper
- **NewSQL** — right idea, wrong hardware bet
- **Accelerators** — a 50-year failure, cloud vendors excepted
- **Blockchain** — "a technology looking for an application"

Parting Lessons

- Never underestimate **marketing for bad products**
- Beware DBMSs from **non-DBMS companies**
- The first five minutes is a technical requirement
- Developers must reach the database directly
- LLMs: real, but **not a replacement for SQL**
- Stop rebuilding buffer pools

Outline

- Data Models: Then, and now
  - History of Data Models (“what comes around…”)
- Languages
  - Overview
  - Datalog (“a survey of research…” and “declarative networking…”)
- Map-reduce and Spark
  - Original MR Abstraction (”mapreduce:” …”)
  - Spark (“resilient distributed datasets…”)
- SystemML: An abstraction for ML
- GraphX: An abstraction for Graphs

Procedural vs Declarative Languages

- Procedural/imperative query languages
  - Support a set of data-oriented operations
  - Usually need to specify the sequence of steps to be taken to get to the output
  - Often map one-to-one with the physical operators that are implemented
    - Large gap between those two ➔ more opportunities to optimize
- Declarative query languages
  - Specify the desired outcome, typically as a function over the inputs
- A different issue than how ”high-level” or abstract the language is
- Most languages today are somewhere in-between
  - SQL is more declarative than procedural

Early Languages

- DL/1 or DBTG were procedural in nature
- Not easy to write complex data processing operations

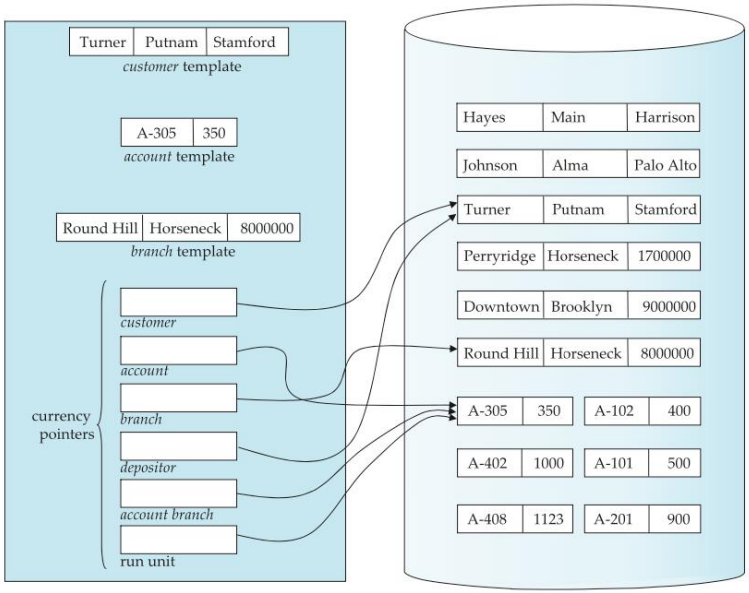


Figure D.20 Program work area.

```
branch.branch_name := "Perryridge";
find any branch using branch_name;
find first account within account_branch;
while DB-status = 0 do
    begin
        find owner within depositor;
        get customer;
        print (customer.customer_name);
        find next account within account_branch;
    end
```

DBTG: CODASYL Language

Relational Algebra

- Original set of operators (select, project, join, union/intersection) proposed by Ted Codd in 1970
  - His "join" operation was somewhat different, but "natural join" same as today
  - Also procedural, but high-level and easy to use

- Six Basic Operations operating on Relations (see 424 slides for more)
  - Select (σ): Unary – select a subset of rows
  - Project (π): Unary – select a subset of columns
  - Set Union and Set Difference: Binary
  - Cartesian Product (×): Binary
  - Rename (ρ)*:* Need to be able to do self-joins

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

r × s:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Inputs and Outputs

- Sets or "multi-sets" of ..
  - Objects → key-value pairs → tuples

o1
o2
o3
o4

Set of objects (e.g., images, JSON documents)

k1	o1
k1	o2
k2	o3
k3	o4

Set of key-value pairs (e.g., pairs of (userid, profile))

o1
o1
o1
o4

Multi-set of objects

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

Set of tuples aka Relation/Table

Operations on Datasets: Select

- Input: Table, Output: Table
- Select only those rows that match the condition
- SQL: “where”
- May be called “match” or “find” or ”filter”

Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

$\sigma_{A=B \wedge D > 5}(r)$

A	B	C	D
α	α	1	7
β	β	23	10

Operations on Datasets: Project

- Input: Table, Output: Table
- Select only those columns that match the condition
- SQL: “select”

Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

$\pi_{A,D}(r)$

A	D
α	7
α	7
β	3
β	10

A	D
α	7
β	3
β	10

Operations on Datasets: Map

- Input: Rows of Objects, Output: Rows of Objects
  - i.e., don't care about what's in a row
- For each row, apply a function
- SQL: "select" can handle this (with UDFs)

Table t

o1
o2
o3
o4

t.map(f)

f(o1)
f(o2)
f(o3)
f(o4)

Operations on Datasets: flatMap

- Has other names (e.g., unwind)
- Input: Rows of Objects, Output: Rows of Objects
- For each row, apply a function that may generate >= 0 rows
- SQL: No easy way to do this

Table t

o1
o2
o3
o4

t.map(f)

f(o1) = [o1', o2', o3']

f(o2) = []

f(o3) = [o4']

f(o4) = [o5'

o1'
o2'
o3'
o4'
o5'

Operations on Datasets: Group(By)

- Input: Collection of (k, v) pairs, Output: Collection of (k, [v]) pairs
  - k = key, v = value
- Group the input rows by the "key"
- Relational Algebra: No support (can't have sets as values)
- SQL: Most implementations support it
  - e.g., postgresql has array aggregates or string aggregates
- Other Names: "nest"

Table t

k1	o1
k1	o2
k2	o3
k3	o4

t.groupByKey()

k1	[o1, o2]
k2	[o3]
k3	[o4]

Operations on Datasets: Group(By) Aggregates

- Input: Collection of (k, v) pairs, Output: Collection of (k, v') pairs, where v' may be different data type
- Also called "reduceByKey" or "aggregateByKey"
- Group values by key, and apply a provided "function" to get a single value
- SQL has a predefined set of functions (SUM, COUNT, MAX, …)

Table t

k1	o1
k1	o2
k2	o3
k3	o4

t.reduceByKey(f)

k1	f(o1, o2)
k2	f(o3)
k3	f(o4)

Operations on Datasets: Group(By) Aggregates

- PostgreSQL (and other systems) support user-defined aggregate functions
  - init(): what's the initial state (e.g., for AVG: (count = 0, sum = 0))
  - update(): modify state given a new value (e.g., for AVG: (count + 1, sum + newval))
  - final(): generate the final aggregate (e.g., for AVG: sum/count)
  - The update operation must be insensitive to the order in which the values are processed
  - i.e., output should be the same if it sees: v1, v2, v3, versus if it sees: v3, v2, v1 in that order
  - Must process tuples sequentially

```
s = init()
s = update(s, v1)
s = update(s, v2)
…
result = final(s)
```

- Another way to do it
  - Provide a binary function that is commutative and distributive
  - Shouldn't matter in which order the objects are processed
  - More "parallelizable"
  - Can generate a (sum, count) pair, but for "average" need another "map"

Any of these are fine

```
result = f(f(f(v1, v2), f(v3, v4)), v5)
result = f(f(f(v1, f(v2, v3)), v4)), v5)
result = f(f(f(f(f(v1, v2), v3), v4), v5)
result = f(f(v1, f(v2, v3)), f(v4, v5))
```

Operations on Datasets: Unnest

- Opposite of “nest” (group by)
- Similar to “flatMap” and “unwind” (in MongoDB)
  - But defined for relational algebra (extended to handle sets as values)
- Useful abstraction to deal with non-1NF data (e.g., JSON which supports arrays)

Table t

k1	[o1, o2]
k2	[o3]
k3	[o4]

unnest

k1	o1
k1	o2
k2	o3
k3	o4

Operations on Datasets: Unnest

SUBJECT	DEGREE
SUBJECT□	
Composition Maths Physics	Science
Composition Statistics Maths	Economics
Composition Maths Chemistry	Science
Composition History Latin Ancient Greek	Arts

UNNEST($_{\text{SUBJECT(SUJBECT')}}$EXAMS)

SUBJECT□	DEGREE
Composition	Science
Maths	Science
Physics	Science
Composition	Economics
Statistics	Economics
Maths	Economics
Composition	Science
Maths	Science
Chemistry	Science
Composition	Classics
History	Classics
Latin	Classics
Ancient Greek	Classics

From: Nest and Unnest operations in nested relations
Georgia Garani
Data Science Journal, May 2008

Operations on Datasets: Set Union/Intersection/Difference

- Input: Two collections, Output: One collection
- SQL support: union/except/intersection
  - Requires collections (tables) to have the same schema
  - Removes duplicates by default (most SQL operations don't remove duplicates)
  - Can use "union all" etc., to preserve duplicates

Relation r, s

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

r U s:

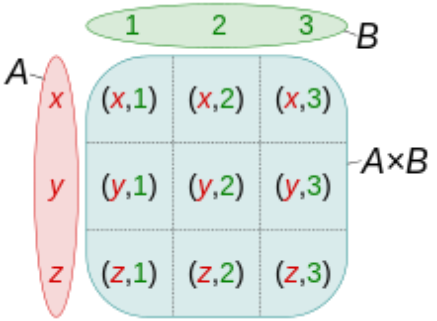
A	B
α	1
α	2
β	1
β	3

r – s:

A	B
α	1
β	1

Operations on Datasets: Cartesian Product

- Input: Two tables, Output: One table
- Note: a “set” product will result in nested output
  - First row would be: ((alpha, 1), (alpha, 10, a))
  - Relation algebra flattens it



Relation r, s

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

r × s:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Operations on Datasets: Joins

- Input: Two tables, Output: One table
- Cartesian product followed by a “select”/“map”
- Many variations of joins used in database literature
  - Note: semi-join and anti-join are technically “select” operations on “r”, not a ”join” operation

Tables: r(A, B), s(B, C)

name	Symbol	SQL Equivalent	RA expression
cross product	$\times$	select * from r, s;	$r \times s$
natural join	$\bowtie$	natural join	$\pi_{r.A,\ r.B,\ s.C}\sigma_{r.B\ =\ s.B}(r\ x\ s)$
theta join	$\bowtie_\theta$	from .. where θ;	$\sigma_\theta(r\ x\ s)$
equi-join	$\bowtie_\theta$ *(theta must be equality)*		
left outer join	$r \ltimes s$	left outer join (with “on”)	(see previous slide)
full outer join	$r \rtimes s$	full outer join (with “on”)	–
(left) semijoin	$r \ltimes s$	none	$\pi_{r.A,\ r.B}(r \bowtie s)$
(left) antijoin	$r \triangleright s$	none	$r - \pi_{r.A,\ r.B}(r \bowtie s)$

Operations on Datasets: Lookup

- Another way to look at a left outer join
- Input: Table, Output: Table
- Augment the input table with data from another table
- Supported by “Excel”, MongoDB, etc.

Table R

A	B	C
a1	b1	c1
a2	b2	c2
a3	b1	c1

Lookup in S using “C”



A	B	C	D	E
a1	b1	c1	d1	e1
a2	b2	c2	d2	e1
a3	b1	c1	d1	e1

C	D	E
c1	d1	e1
c2	d2	e1
c3	d2	e3

Table S

C must be a “key” for S,
o/w not well-defined

Operations on Datasets: Pivot

- Flip rows and columns
- No SQL equivalent, although supported by many systems (e.g., CROSSTAB in PostgreSQL)
- Usually used in conjunction with aggregates (so that the number of rows is small)

Table R

A	B	C
a1	b1	c1
a2	b2	c2
a3	b1	c1

pivot

a1	a2	a3
b1	b2	b1
c1	c2	c1

Operations on Datasets

- Sorting and ordering
- Ranking (sparse vs dense rank)
- Distinct (duplicate elimination)
- Sample: generate a random sample
- Data Cubes
  - Allows aggregating on multiple attributes simultaneously

Recap

- Many data management systems view data as collection/multiset of tuples or objects or (key, value) pairs
- A common set of operations supported by most
  - Some Unary, Some Binary (or more generally, n-ary)
- Language constructs often map one-to-one to physical operators, but not always
  - e.g., SQL JOIN is a n-ary operator, that maps to a sequence of binary JOINs
  - Recent work on trying to do n-ary joins directly (asymptotically better in some cases)
- More declarative a language ➔ More opportunities to optimize
  - e.g., Pandas (Python Library), MongoDB, Apache Spark RDD interface, etc, not declarative even though pretty high-level
  - However, physical operators themselves can be heavily optimized, especially in parallel settings

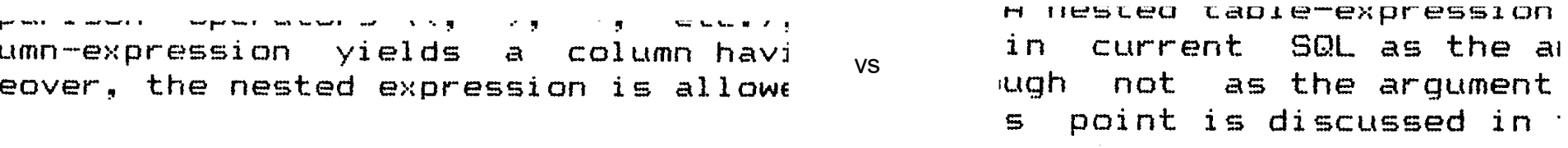
SQL

- Originally SEQUEL: A Structured English Query Language (1974), developed at IBM for System R
- Commercial implementations in Oracle and DB2 in late 70's, early 80's
- Standardization by ANSI and ISO started in 1996

- Very similar to Relational Algebra in the basic operators it supports
  - Except for GROUP BYs and AGGREGATEs (among basic constructs), and some Set Operations (e.g., NOT IN, ALL)
- Modern implementations support many additional constructs
  - Window and Partitioning Functions
  - Recursion
  - Triggers
- Skim through 424 Slides

Some Criticisms of SQL

- From: " A critique of SQL", Date, 1984
- e.g., can't use a table name as a table expression



- In general, a lot of inconsistencies in what can be used where
- Aggregates don't have a natural formalism

- Some of those criticisms fixed since then, but many are fundamental to the language

Impedance Mismatch

- Recognized fairly early on: ”Some High Level Language Constructs for Data of Type Relation”; Schmidt, 1976
- Proposed an extension to Pascal/R to include a Relation as a basic data type

```
type  erectype = record enr, estatus:integer; ename:string end;
      ereltype = relation ⟨enr⟩ of erectype;
      trectype = record tenr, tcnr, ttime:integer;
                        tday, troom:string end;
      treltype = relation ⟨tenr, tcnr, tday⟩ of trectype;
      crectype = record cnr, clevel:integer; cname:string end;
      creltype = relation ⟨cnr⟩ of crectype;
var   employees, result3, result4, result5, result6:ereltype;
      timetable:treltype;

      courses:creltype;
begin .
      .
      .
      result3 := [each erec in employees:erec.estatus = 2];
      result4 := [each erec in employees:some trec in timetable ((erec.enr = trec.tenr) and
                        (trec.tday = 'friday'))];
      result5 := [each erec in employees:all trec in timetable (erec.enr ≠ trec.tenr)];
      result6 := [each erec in employees:all trec in timetable ((erec.enr ≠ trec.tenr) or
                        some crec in courses ((trec.tcnr = crec.cnr) and (crec.clevel = 1)))]
end.
```

Impedance Mismatch

- Also led to the work on Object-oriented Databases, and (later) Object-Relational Mapping (ORM) frameworks
- Language INtegrated Query (LINQ) Framework
  - A declarative component of the .NET OOPLs (C#, Visual Basic, F#)
  - Allows querying and manipulating collections of objects using SQL-style syntax

```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

```
// SQL-style syntax to join two input sets:
// scoreTriples and staticRank
var adjustedScoreTriples =
  from d in scoreTriples
  join r in staticRank on d.docID equals r.key
  select new QueryScoreDocIDTriple(d, r);
var rankedQueries =
  from s in adjustedScoreTriples
  group s by s.query into g
  select TakeTopQueryResults(g);

// Object-oriented syntax for the above join
var adjustedScoreTriples =
  scoreTriples.Join(staticRank,
    d => d.docID, r => r.key,
    (d, r) => new QueryScoreDocIDTriple(d, r));
var groupedQueries =
  adjustedScoreTriples.GroupBy(s => s.query);
var rankedQueries =
  groupedQueries.Select(
    g => TakeTopQueryResults(g));
```

Figure 3: *A program fragment illustrating two ways of expressing the same operation. The first uses LINQ's declarative syntax, and the second uses object-oriented interfaces. Statements such as* `r => r.key` *use C#'s syntax for anonymous lambda expressions.*

From the DryadLINQ Paper, a distributed implementation of LINQ

Very similar to Apache Spark

Impedance Mismatch

- Also led to the work on Object-oriented Databases, and (later) Object-Relational Mapping (ORM) frameworks
- Language INtegrated Query (LINQ) Framework
  - A declarative component of the .NET OOPLs (C#, Visual Basic, F#)
  - Allows querying and manipulating collections of objects using SQL-style syntax
- Today, many programming languages have support for list comprehensions, dictionaries, and features like that
  - With libraries to make it easy to load and store data
  - For example, Pandas for Python, Libraries to read/write Parquet/Avro files, etc.

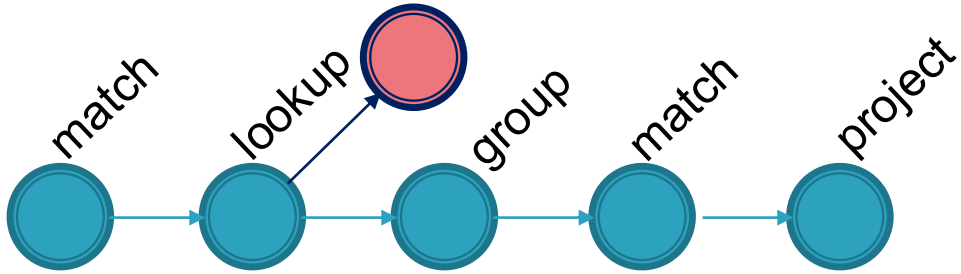
MongoDB Query Language (MQL)

- Input = collections, output = collections
  - Very similar to Spark
- Three main types of queries in the query language
  - Retrieval: Restricted SELECT-WHERE-ORDER BY-LIMIT type queries
  - Aggregation: A bit of a misnomer; a general pipeline of operators
    - Can capture Retrieval as a special case
    - But worth understanding Retrieval queries first…
  - Updates
- All queries are invoked as
  - db.collection.operation1(…).operation2(…)…
    - collection: name of collection
  - Unlike SQL which lists many tables in a FROM clause, MQL is centered around manipulating a single collection (like Spark)

Syntax somewhat different when called from within Python3 (using pymongo)

Aggregation Pipelines

- Composed of a linear *pipeline* of *stages*
- Each stage corresponds to one of:
  - match
  - project
  - sort/limit
  - group
  - unwind
  - lookup
  - … lots more!!
- Each stage manipulates the existing collection in some way



- Syntax:

```
db.collection.aggregate ( [
 { $stage1Op: { } },
 { $stage2Op: { } },
  …
 { $stageNOp: { } }
] )
```

Grouping (with match/sort) Simple Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
```

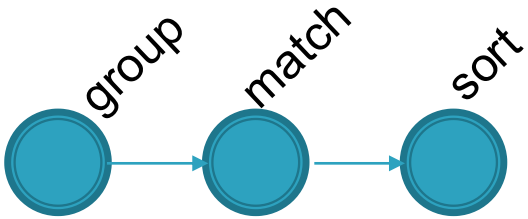
Find states with population > 15M, sort by decending order

```
db.zips.aggregate( [
{ $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
{ $match: { totalPop: { $gte: 15000000 } } },
{ $sort : { totalPop : -1 } }
 ] )
```

GROUP BY

AGGS.

match after group = HAVING




```
{ "_id" : "CA", "totalPop" : 29754890 }
{ "_id" : "NY", "totalPop" : 17990402 }
{ "_id" : "TX", "totalPop" : 16984601 }
…
```

Q: what would the SQL query for this be?

SELECT state AS id, **SUM**(pop) **AS** totalPop
FROM zips
GROUP BY state
HAVING totalPop >= 15000000
ORDER BY totalPop **DESCENDING**

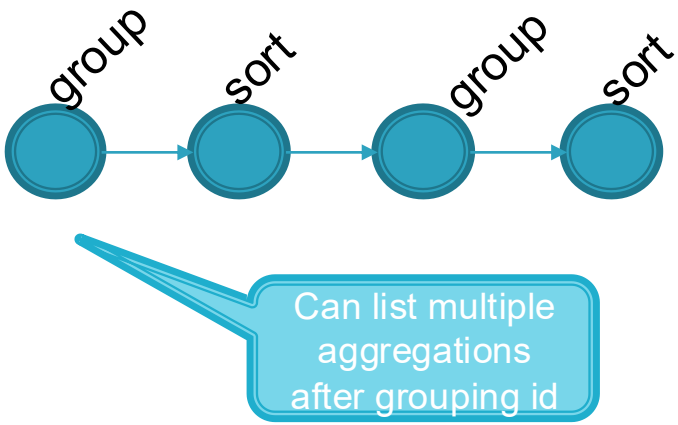
Syntax somewhat different when called from within Python3 (using pymongo)

Multiple Agg. Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
```

Find, for every state, the biggest city and its population

```
aggregate( [
{ $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
{ $sort: { pop: -1 } },
{ $group: { _id : "$_id.state", bigCity: { $first: "$_id.city" }, bigPop: { $first: "$pop" } } },
{ $sort : {bigPop : -1} }
] )
```



Approach:

- Group by pair of city and state, and compute population per city
- Order by population descending
- Group by state, and find first city and population per group (i.e., the highest population city)
- Order by population descending

```
{ ”_id" : "IL", "bigCity" : "CHICAGO", "bigPop" : 2452177 }
{ "_id" : "NY", "bigCity" : "BROOKLYN", "bigPop" : 2300504 }
{ "_id" : "CA", "bigCity" : "LOS ANGELES", "bigPop" : 2102295 }
{ "_id" : "TX", "bigCity" : "HOUSTON", "bigPop" : 2095918 }
{ "_id" : "PA", "bigCity" : "PHILADELPHIA", "bigPop" : 1610956 }
{ "_id" : "MI", "bigCity" : "DETROIT", "bigPop" : 963243 }
...
```

Syntax somewhat different when called from within Python3 (using pymongo)

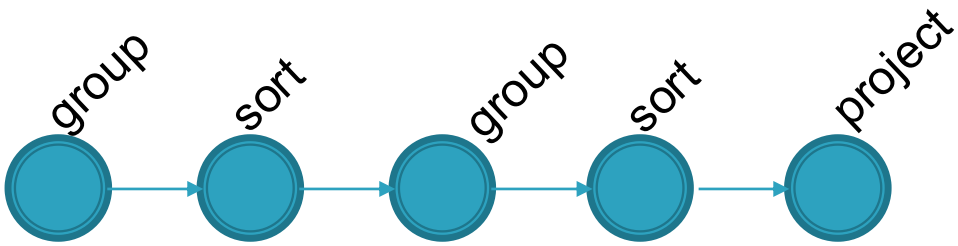
Multiple Agg. with Vanilla Projection Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
```

If we only want to keep the state and city …

```
aggregate( [
{ $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
{ $sort: { pop: -1 } },
{ $group: { _id : "$_id.state", bigCity: { $first: "$_id.city" }, bigPop: { $first: "$pop" } } },
{ $sort : {bigPop : -1} }
{ $project : {bigPop : 0} }
] )
```

```
{ "_id" : "IL", "bigCity" : "CHICAGO" }
{ "_id" : "NY", "bigCity" : "BROOKLYN" }
{ "_id" : "CA", "bigCity" : "LOS ANGELES" }
{ "_id" : "TX", "bigCity" : "HOUSTON" }
{ "_id" : "PA", "bigCity" : "PHILADELPHIA" }
…
```



Syntax somewhat different when called from within Python3 (using pymongo)

Multiple Agg. with Adv. Projection Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
```

If we wanted to nest the name of the city and population into a nested doc

Can construct new nested documents in output, unlike vanilla projection

```
aggregate( [
{ $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
{ $sort: { pop: -1 } },
{ $group: { _id : "$_id.state", bigCity: { $first: "$_id.city" }, bigPop: { $first: "$pop" } } },
{ $sort : {bigPop : -1} },
{ $project : { _id : 0, state : "$_id", bigCityDeets: { name: "$bigCity", pop: "$bigPop" } } }
] )
```

```
{ "state" : "IL", "bigCityDeets" : { "name" : "CHICAGO", "pop" : 2452177 } }
{ "state" : "NY", "bigCityDeets" : { "name" : "BROOKLYN", "pop" : 2300504 } }
{ "state" : "CA", "bigCityDeets" : { "name" : "LOS ANGELES", "pop" : 2102295 } }
{ "state" : "TX", "bigCityDeets" : { "name" : "HOUSTON", "pop" : 2095918 } }
{ "state" : "PA", "bigCityDeets" : { "name" : "PHILADELPHIA", "pop" : 1610956 } }
...
```

Syntax somewhat different when called from within Python3 (using pymongo)

Unwind: A Common Template

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

Q: Imagine if we want to find sum of qtys across items. How would we do this?

A common recipe in MQL queries is to *unwind* and then *group by*

```
aggregate( [
        { $unwind : "$instock" },
        { $group : {_id : "$item", totalqty : {$sum : "$instock.qty"}}}
] )
```

```
{ "_id" : "notebook", "totalqty" : 5 }
{ "_id" : "postcard", "totalqty" : 50 }
{ "_id" : "journal", "totalqty" : 20 }
{ "_id" : "planner", "totalqty" : 45 }
{ "_id" : "paper", "totalqty" : 75 }
```

Syntax somewhat different when called from within Python3 (using pymongo)

Looking Up Other Collections

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

```
{ $lookup: {
        from: <collection to join>,
        localField: <referencing field>,
        foreignField: <referenced field>,
        as: <output array field>
    } }
```

Conceptually, for each document

- find documents in other coll that join (equijoin)
  - local field must match foreign field
- place each of them in an array

Thus, a left outer equi-join, with the join results stored in an array

Straightforward, but kinda gross. Let's see…

Say, for each item, I want to find other items located in the same location = self-join

```
db.inventory.aggregate( [
    { $lookup : {from : "inventory", localField: "instock.loc", foreignField:
    "instock.loc", as:"otheritems"}},
    { $project : {_id : 0, tags : 0, dim : 0}}
] )
```

```
{ "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" :
15 } ], "otheritems" : [
    { "_id" : ObjectId("5fb6f9605f0594e0227d3c24"), "item" : "journal",
    "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" :
    [ "blank", "red" ], "dim" : [ 14, 21 ] },
    { "_id" : ObjectId("5fb6f9605f0594e0227d3c25"), "item" :
    "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red",
    "blank" ], "dim" : [ 14, 21 ] },
    { "_id" : ObjectId("5fb6f9605f0594e0227d3c26"), "item" : "paper",
    "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags"
    : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] },
    …
    ] }
```

And many other records!

Syntax somewhat different when called from within Python3 (using pymongo)

Outline

- Data Models: Then, and now
  - History of Data Models ("what comes around…")
  - A data model for Key-value Stores ("a co-relational model..")
- Languages
  - Overview
  - Datalog ("a survey of research…" and "declarative networking…")
- Map-reduce and Spark
  - Original MR Abstraction ("mapreduce:" …")
  - Spark ("resilient distributed datasets…")
- SystemML: An abstraction for ML
- GraphX: An abstraction for Graphs

Tuple Relational Calculus

- Non-procedural Language (unlike RA)
- Basic Query: all tuples such that P(t) is true

$$\{t \mid P(t)\}$$

- Example: Find instructors with department in the Watson Building

$$\{t \mid \exists\, s \in \mathit{instructor}\,(t[\mathit{name}] = s[\mathit{name}] \wedge \exists\, u \in \mathit{department}\,(u[\mathit{dept_name}] = s[\mathit{dept_name}] \wedge u[\mathit{building}] = \text{“Watson”}))\}$$

- Find students who have taken all courses offered by Biology

$$\{t \mid \exists\, r \in \mathit{student}\ (r[\mathit{ID}] = t[\mathit{ID}]) \wedge (\forall\, u \in \mathit{course}\,(u[\mathit{dept_name}] = \text{“Biology”} \Rightarrow \exists\, s \in \mathit{takes}\,(t[\mathit{ID}] = s[\mathit{ID}] \wedge s[\mathit{course_id}] = u[\mathit{course_id}]))\}$$

Datalog

- From Online Chapter at: https://db-book.io
- Extensional ”Facts”
  - Map to tuples in relations
- “Rules”
  - Allow inferring additional ‘intentional’ facts
  - Can be thought of as defining “views”

- Example: *account* is extensional, and *v1* allows inferring additional facts

$$v1(A, B) \text{ :- } account(A, \text{“Perryridge”}, B), B > 700$$

means:

for all A, B
if $(A, \text{“Perryridge”}, B) \in account$ and $B > 700$
then $(A, B) \in v1$

Datalog

- From Online Chapter at: https://db-book.io
- Example: *account* is extensional, and *v1* allows inferring additional facts

$$v1(A, B) \text{ :- } account(A, \text{“Perryridge”}, B), B > 700$$

- Writing queries?

$$?\, v1(A, B), B > 800 \quad \Rightarrow \quad (\text{A-201}, 900)$$

- Multiple rules typically used for the same view

$$interest_rate(A, 5) \text{ :- } account(A, N, B), B < 10000$$
$$interest_rate(A, 6) \text{ :- } account(A, N, B), B >= 10000$$

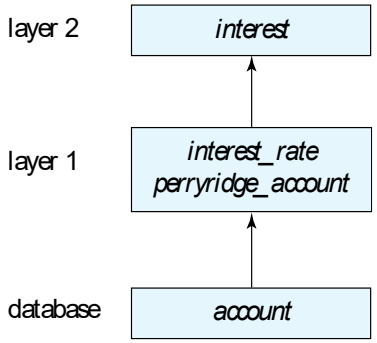
- Can use Negation

$$c(N) \text{ :- } depositor(N,A), \text{not } is_borrower(N)$$
$$is_borrower(N) \text{ :- } borrower(N, L)$$

Datalog

- For non-recursive queries, the semantics are pretty straightforward

```
interest(A, I) :– account(A, "Perryridge", B),
                  interest_rate(A, R), I = B * R/100
interest_rate(A, 5) :– account(A, N, B), B < 10000
interest_rate(A, 6) :– account(A, N, B), B >= 10000
perryridge_account(X, Y) :– account(X, "Perryridge", Y)
```



- More complex for recursive queries
  - Assume we have a single relation: parent(child_id, parent_id)
  - The following program gets ancestors

```
ancestor(A, B) :- parent(B, A)
ancestor(A, B) :- ancestor(A, C), parent(B, C)
```

Datalog: Safety

- Possible to write rules that generate infinite answers

$$gt(X, Y) \text{ :- } X > Y$$
$$notLinLoan(L, B, A) \text{ :- not } loan(L, B, A)$$

- Datalog programs must satisfy safety conditions:
  - Every variable in the head, must appear in a non-arithmetic positive literal in the body
  - Every variable in a negative literal in the body must appear in some positive literal in the body

- For non-recursive program, this guarantees finite results as long as the database relations are finite
- Can relax the rules somewhat:

$$p(A) \text{ :- } q(B), A = B + 1$$

Datalog: Mapping to RA

- Select

  $v1(A, B) \text{ :- } account(A, \text{“Perryridge”}, B), B > 700$

- Project

  $query(A) \text{ :- } account(A, N, B)$

- Cartesian Product

  $query(X_1, X_2, \ldots, X_n, Y_1, Y_2, \ldots, Y_m) \text{ :- } r_1(X_1, X_2, \ldots, X_n), r_2(Y_1, Y_2, \ldots, Y_m)$

- Union

  $query(X_1, X_2, \ldots, X_n) \text{ :- } r_1(X_1, X_2, \ldots, X_n)$
  $query(X_1, X_2, \ldots, X_n) \text{ :- } r_2(X_1, X_2, \ldots, X_n)$

- Set Difference

  $query(X_1, X_2, \ldots, X_n) \text{ :- } r_1(X_1, X_2, \ldots, X_n), \text{ not } r_2(X_1, X_2, \ldots, X_n)$

- Extensions exist for aggregates as well

Datalog: Bottom-up vs Top-down Evaluation

- Prolog uses a top-down evaluation approach (by default)
  - Start with query as goal and use the rules to create more goals until you get to facts

```
fibo_td(0,0).
fibo_td(1,1)
fibo_td(N,F):-
  N>1, N1 is N-1, N2 is N-2,
  fibo_td(N1,F1), fibo_td(N2,F2),
  F is F1+F2.
```

  - If we are asked to compute fibo_td(10, F) – we would expand the last rule, and compute fibo_td(9, ?) and fibo_td(8, ?) first
  - Need to use memo-ization in order to avoid exponential runtime

- Not a good approach for Datalog
  - Efficient evaluation requires use of set-at-a-time processing, i.e., start with base facts and generate more facts using the rules
  - Drawback: may generate facts that are not needed

Datalog: Bottom-up Evaluation

- Standard Fixpoint algorithm
  - Start with all the facts (initially the base relations)
  - Infer new facts using those and the provided rules
  - Repeat until no more facts are inferred
- Sometimes called “naïve” evaluation
- Semi-Naïve Evaluation
  - Keep track of which new “facts” were inferred in iteration N - 1
  - In iteration N: only consider those rules as firing that include at least one of those facts
- Works for both recursive and non-recursive programs
  - Bounded depth for non-recursive programs based on the query

Datalog: Bottom-up Evaluation

- For “safe” Datalog programs, this will stop at some point assuming no “negative” literals
- With negative literals, previous inferences may be invalidated
  - e.g., q(X, Y) :- not R(Y, X), Y = 10, X = 5
  - If R(10, 5) doesn’t exist, we can infer q(5, 10)
  - However in a later iteration, we may get: R(10, 5)
- Note: SQL originally did not support recursion – so no way to do “transitive closure” – SQL 99 added support
- We will discuss some other optimization techniques (e.g., magic sets) later

Declarative Networking using Datalog

- Challenging to design new network protocols to handle rapidly evolving landscape
  - Correctness particularly an issue
  - Hard to optimize when the bottlenecks change
- Proposed solution
  - Model the distributed state across the routers/machines as ”tables”
  - Use a recursive query language to define derived data, constraints, etc.
- Long line of work starting with this early paper in SIGMOD 2006
- More recent work on distributed programming in general by Hellerstein et al.

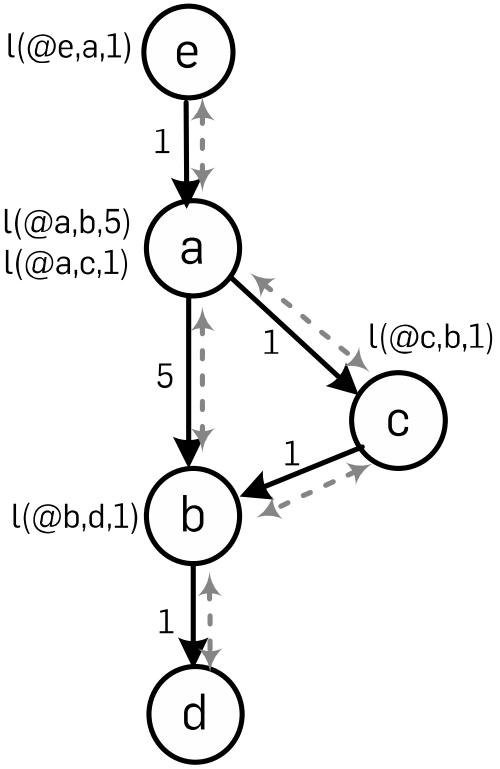
Network Datalog: Example

- Base “extensional” relation: link(Src, Dest, Cost)
  - Stored in a distributed manner across all nodes
- Four rules:
  - sp1 and sp2 define a ”path” in the network recursively
  - sp3 and sp4: an aggregate function to compute minimum-cost path
- @ used to specify where the derived fact should be stored

```
sp1 path(@Src,Dest,Path,Cost) :- link(@Src,Dest,Cost),
    Path=f_init(Src,Dest).
sp2 path(@Src,Dest,Path,Cost) :- link(@Src,Nxt,Cost1),
    path(@Nxt,Dest,Path2,Cost2), Cost=Cost1+Cost2,
    Path=f_concatPath(Src,Path2).
sp3 spCost(@Src,Dest,min<Cost>) :- path(@Src,Dest,Path,Cost).
sp4 shortestPath(@Src,Dest,Path,Cost) :-
    spCost(@Src, Dest,Cost), path(@Src,Dest,Path,Cost).
Query shortestPath(@Src,Dest,Path,Cost).
```

Network Datalog: Example Execution

- In each iteration, nodes do their local computations and pass their state to their neighbors



Initially

Network Datalog: Extensions

- Need to handle limitations of the underlying network

- Link-restricted rules:
  - Not all nodes can talk directly to all nodes for execution of the program
  - Only allow rules where there is a direct link between the two nodes that contain the data required for any predicate

```
p(@Dest,...) :- link(@Src,Dest...),p1(@Src,...),
                p2(@Src,...),..., pn(@Src,...).
```

- Soft state storage:
  - Network protocols data typically has a TTL (time-to-live)
  - Introduce a new keyword: materialized
  - Adds some complications in formal semantics

Outline

- Data Models: Then, and now
  - History of Data Models (“what comes around…”)
  - A data model for Key-value Stores (“a co-relational model..”)
- Languages
  - Overview
  - Datalog (“a survey of research…” and “declarative networking…”)
- Map-reduce and Spark
  - Original MR Abstraction (”mapreduce:” …”)
  - Spark (“resilient distributed datasets…”)
- SystemML: An abstraction for ML
- GraphX: An abstraction for Graphs

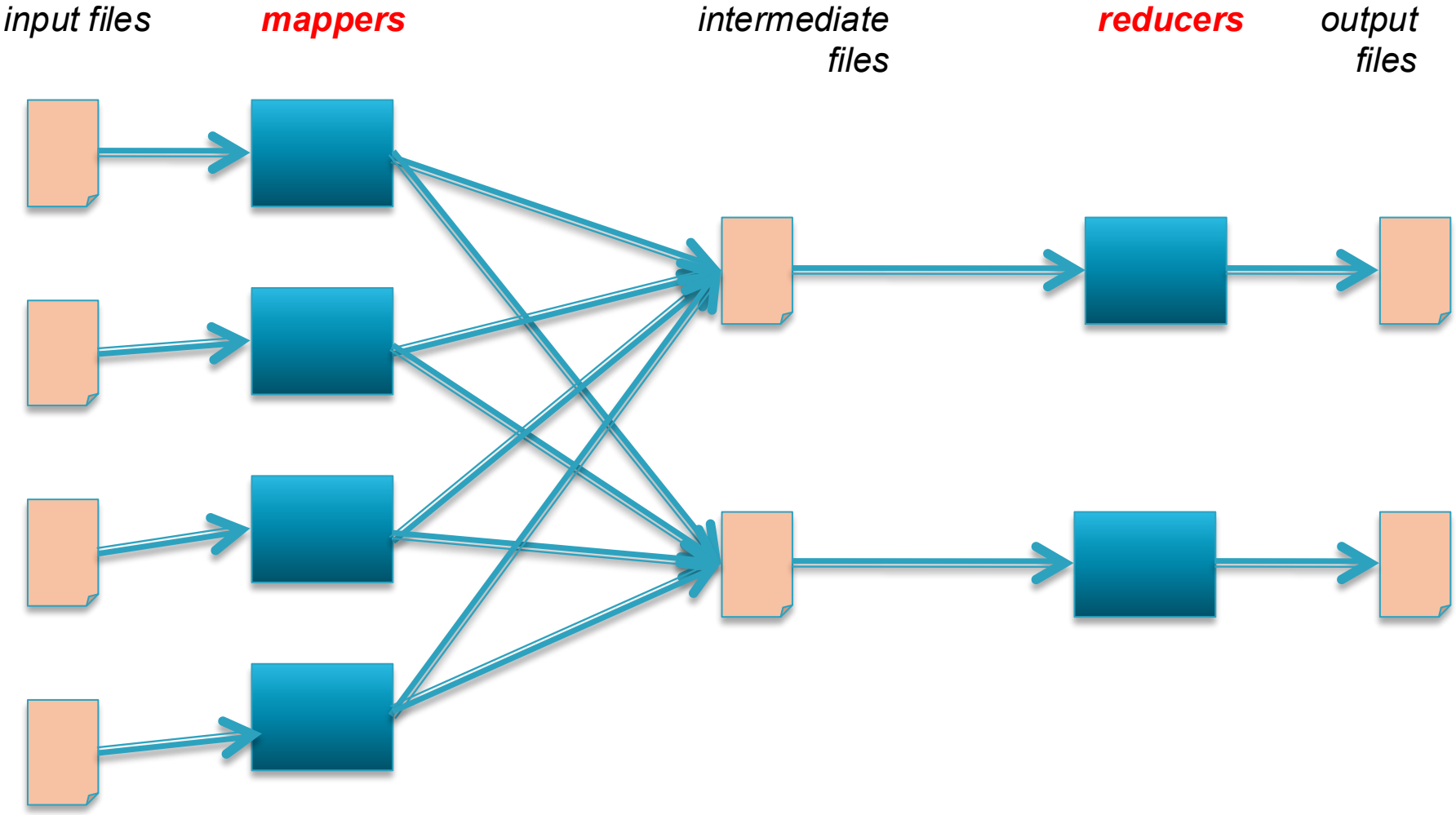
Map Reduce; 2003

- For parallel, fault-tolerant computation over large volumes of data
- Just two operators: “map” and “reduce”
  - Map more like “flatMap” – can produce multiple outputs per input
  - “reduce” == “reduceByKey” – operated on key-value pairs

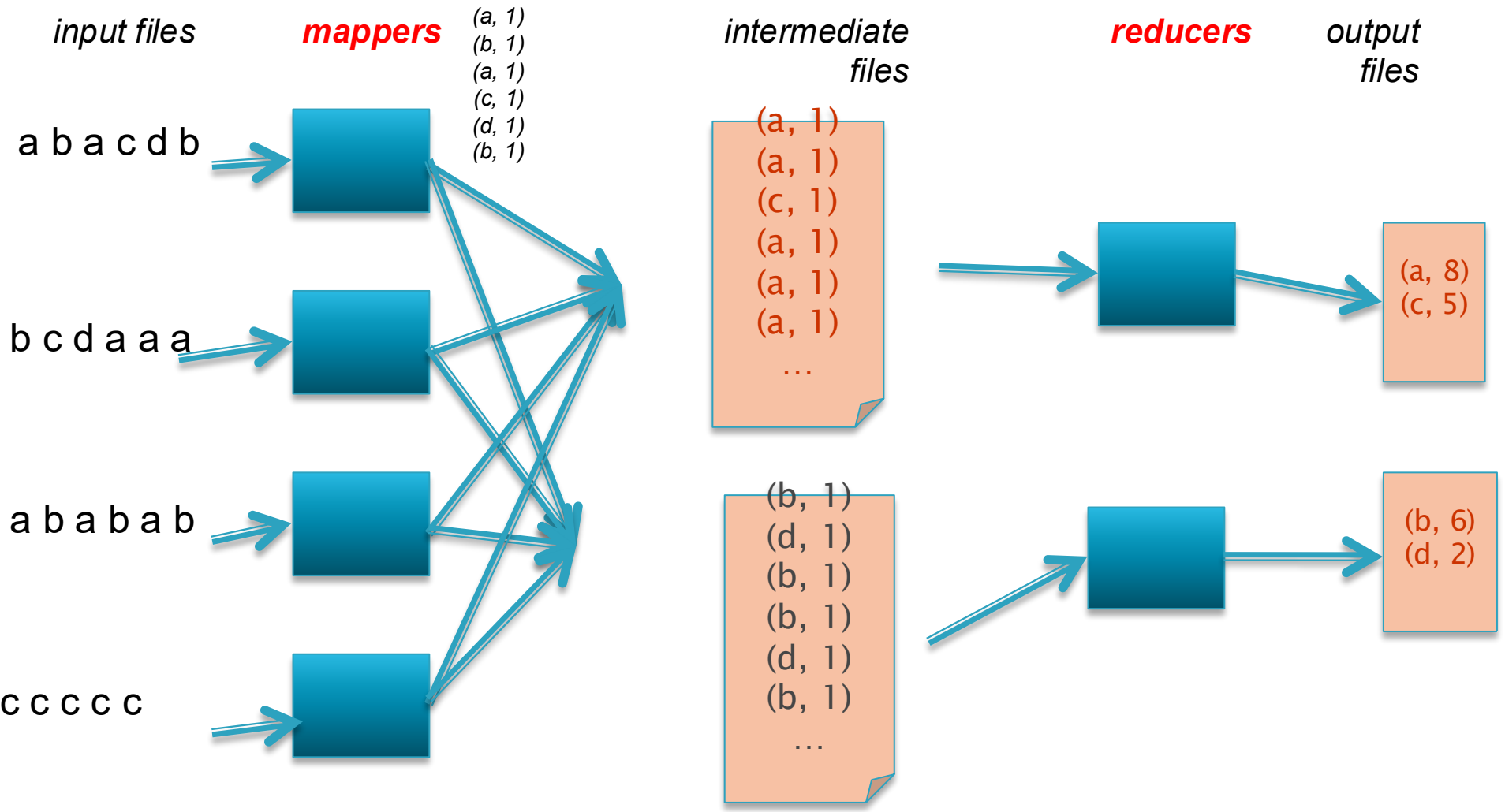
```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

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

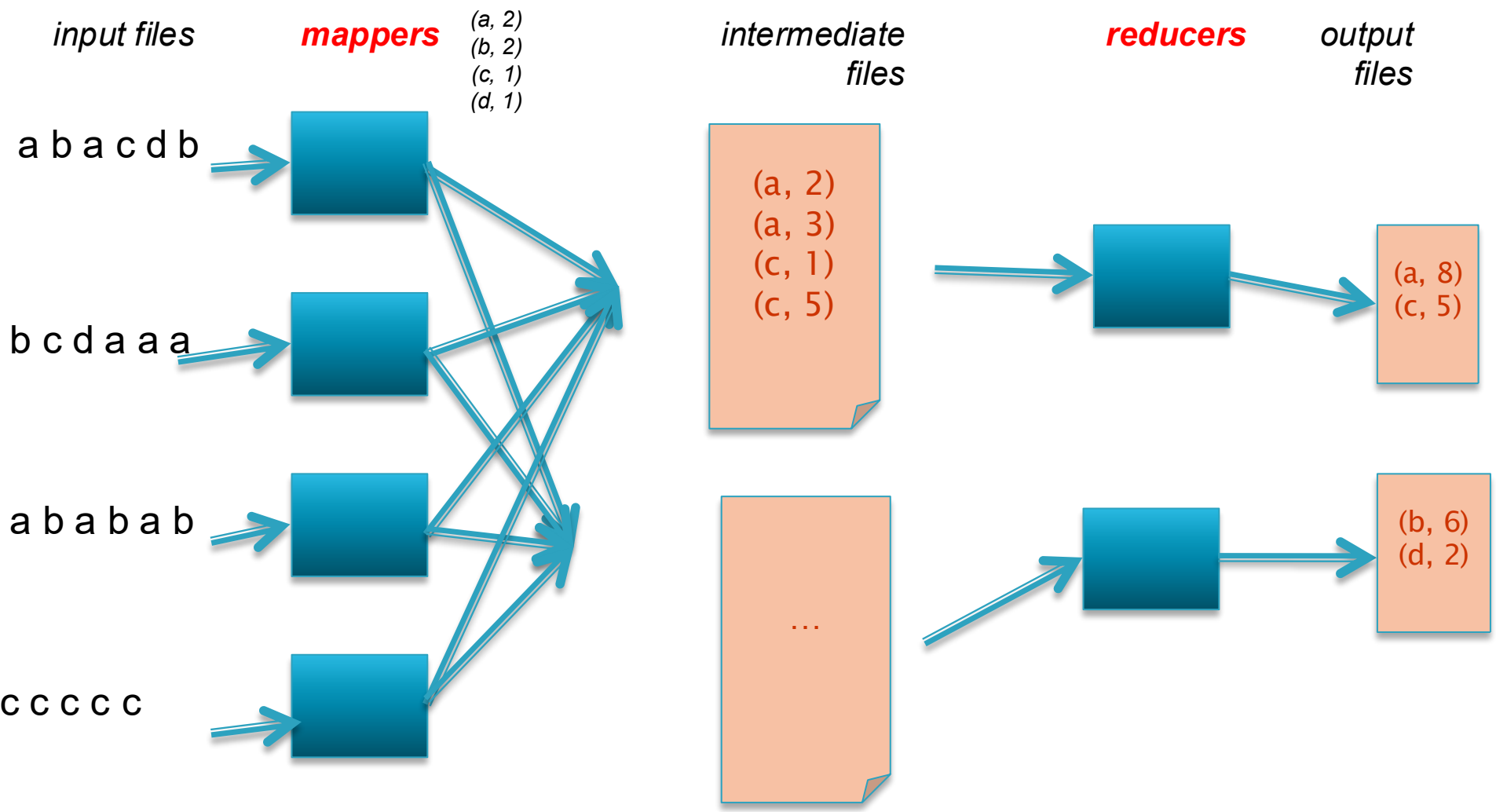
MapReduce Framework



MapReduce Framework: Word Count



More Efficient Word Count



Called “mapper-side” combiner
Partial aggregation in DBMS terms

Map Reduce; 2003

- For parallel, fault-tolerant computation over large volumes of data
- Just two operators: “map” and “reduce”
  - Map more like “flatMap” – can produce multiple outputs per input
  - “reduce” == “reduceByKey” – operated on key-value pairs
- Each operator is ”embarrassingly” (“infinitely”) parallelizable
  - Map can be done in parallel on each input
- Data written out to disk after each map or reduce
  - For fault-tolerance
  - If a machine fails, restart the computation on another machine with the same input files
- Many optimizations to handle skew, etc.

Map Reduce; 2003

- Was used in Google (at that time) for:
  - Large-scale machine learning problems
  - Clustering problems for Google News etc
  - Generating summary reports
  - Large-scale Graph Computations
  - Extract-Transform-Load (ETL) tasks
- Also replaced original tools for large-scale indexing
  - i.e., for generating the inverted indexes
  - runs as a sequence of 5 to 10 MapReduce opeartions

Map Reduce vs RDBMS

- Limited functionality, but no RDMBS/data warehouse could have handled those kinds of tasks
  - Not fault-tolerant at the scale
  - Most of the data not tabular or relation – SQL not a good fit
    - Need flexible or no schemas
    - User-defined functions can help but hard to use back then
  - Loading the data into databases not feasible
    - Much of the analysis is one-time
  - Cost prohibitive (Distributed File Systems much cheaper)
- Mapreduce: A Major Step Backwards; DeWitt and Stonebraker; 2007
- See the later CACM papers by both camps

Post-MapReduce Systems

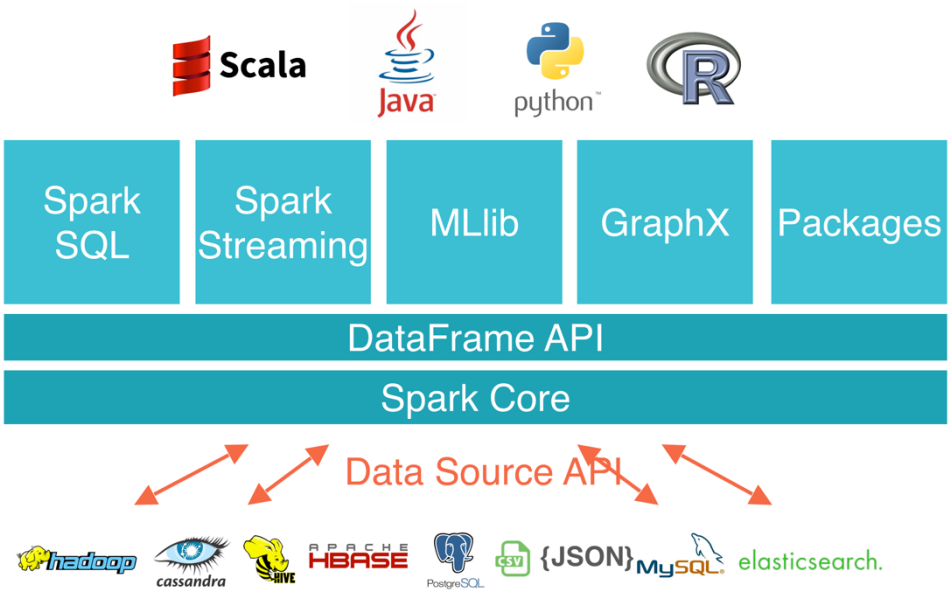
- Yahoo! open-sourced the Hadoop MapReduce; 2006
  - Including other tools like Zookeeper, HDFS, etc.
  - Many inter-operable modules built around this sinc then
- Soon afterwards: Dryad (MSFT), Hive (FB), Pig (Yahoo)
  - Most supported higher-lever interfaces: Hive and Pig more like SQL, whereas Dryad supported something like LINQ
- Latest generation of systems: Spark, F1, Impala, Tez, Naiad, Flink, AsterixDB, Drill, etc…
  - Higher-level query languages like SQL
  - More advanced execution strategies
  - Indexes, query optimization, etc.
  - Support for streaming, ML, graphs, etc.

Outline

- Data Models: Then, and now
  - History of Data Models ("what comes around…")
  - A data model for Key-value Stores ("a co-relational model..")
- Languages
  - Overview
  - Datalog ("a survey of research…" and "declarative networking…")
- Map-reduce and Spark
  - Original MR Abstraction ("mapreduce:" …")
  - Spark ("resilient distributed datasets…")
- SystemML: An abstraction for ML
- GraphX: An abstraction for Graphs

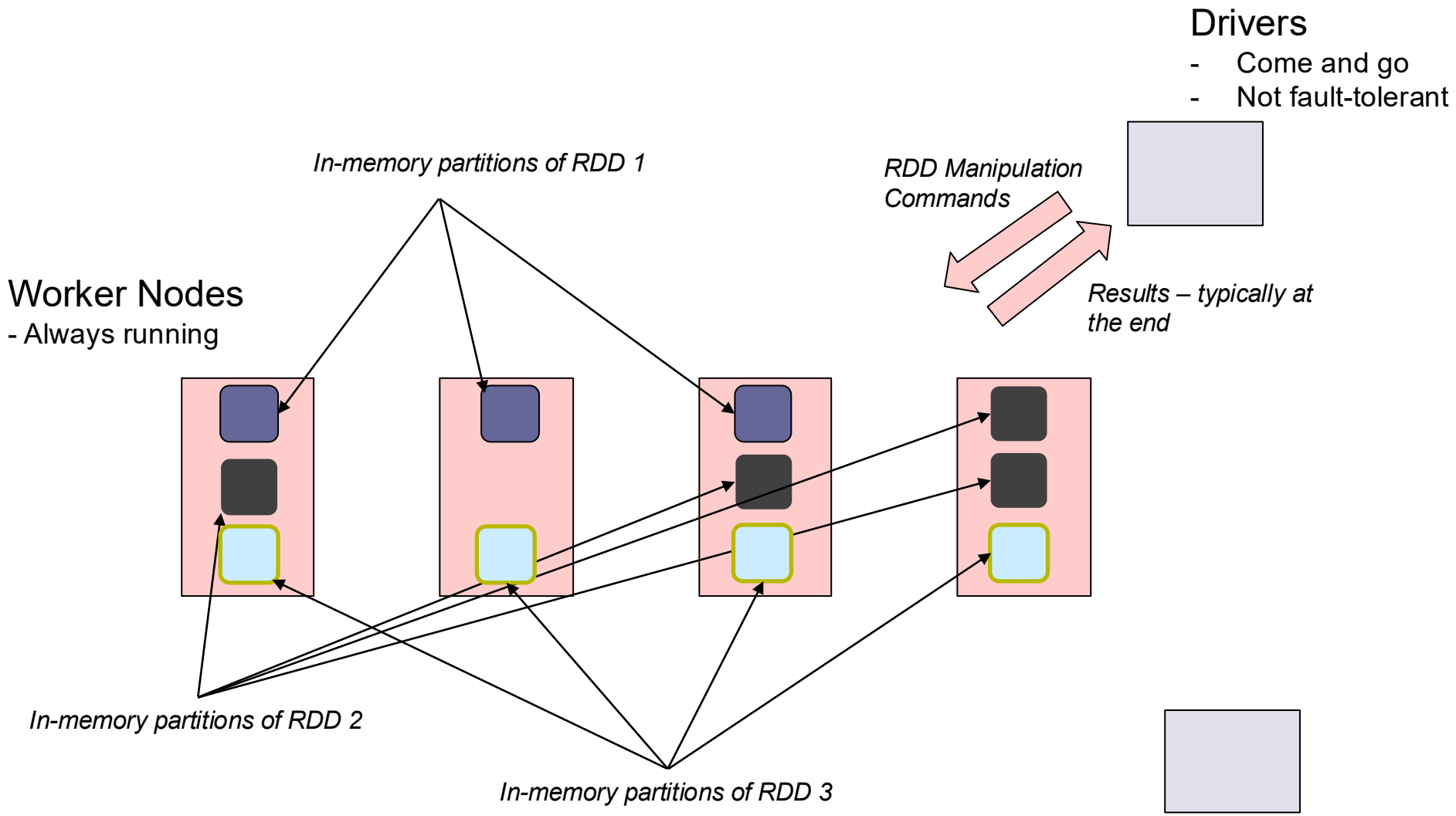
Spark

- Open-source, distributed cluster computing framework
- Much better performance than Hadoop MapReduce through in-memory caching and pipelining
- Originally provided a low-level RDD-centric API, but today, most of the use is through the “Dataframes” (i.e., relations) API
  - ★ Dataframes support relational operations like Joins, Aggregates, etc.



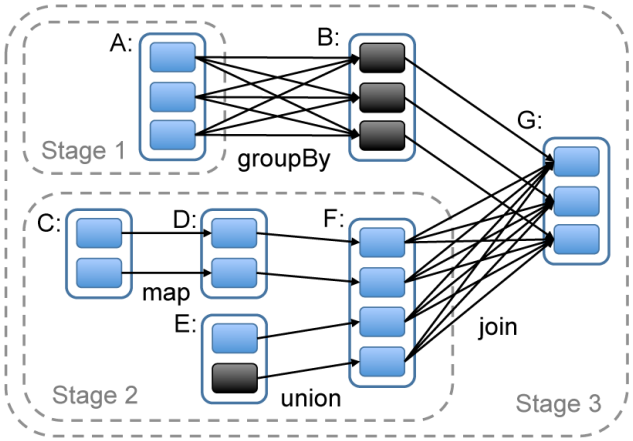
Resilient Distributed Dataset (RDD)

- **RDD =** Collection of records stored across multiple machines in-memory



Spark

- Why “Resilient”?
  - Can survive the failure of a worker node
  - Spark maintains a “lineage graph” of how each RDD partition was created
  - If a worker node fails, the partitions are recreated from its inputs
  - Only a small set of well-defined operations are permitted on the RDDs
    - But the operations usually take in arbitrary ”map” and “reduce” functions



- Fault tolerance for the “driver” is trickier
  - Drivers have arbitrary logic (cf., the programs you are writing)
  - In some cases (e.g., Spark Streaming), you can do fault tolerance
  - But in general, driver failure requires a restart

Example Spark Program

Initialize RDD by reading the textFile and partitioning
If textFile stored on HDFS, it is already partitioned – just read each partition as a separate RDD partition

Split each line into words, creating an RDD of words

For each word, output (word, 1), creating a new RDD

Do a group-by SUM aggregate to count the number of times each word appears

Driver

```
from pyspark import SparkContext

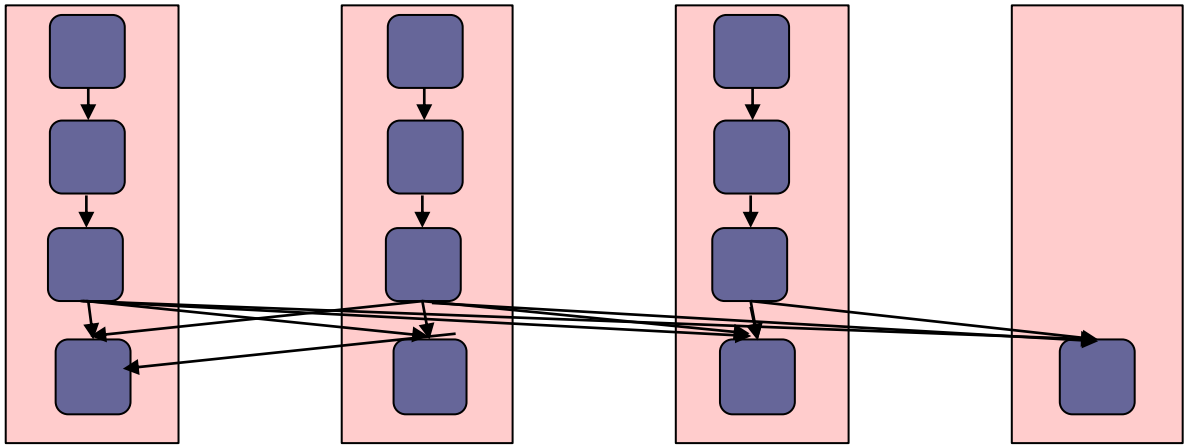
sc = SparkContext("local", "Simple App")

textFile = sc.textFile("README.md")

counts = textFile
        .flatMap(lambda line: line.split(" "))
        .map(lambda word: (word, 1))
        .reduceByKey(lambda a, b: a + b)

print(counts.take(100))
```

Retrieve 100 of the values in the final RDD



RDD Operations

Transformations

The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R) and pair RDD functions doc (Scala, Java) for details.

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numPartitions*])	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks.
reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numPartitions*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean

Actions

The following table lists some of the common actions supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R)

and pair RDD functions doc (Scala, Java) for details.

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`.
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the `foreach()` may result in undefined behavior. See Understanding closures for more details.

Dataframes Example

```
def basic_df_example(spark):
    # $example on:create_df$
    # spark is an existing SparkSession
    df = spark.read.json("examples/src/main/resources/people.json")
    # Displays the content of the DataFrame to stdout
    df.show()
    # +----+-------+
    # | age|   name|
    # +----+-------+
    # |null|Michael|
    # |  30|   Andy|
    # |  19| Justin|
    # +----+-------+
    # $example off:create_df$


    # $example on:untyped_ops$
    # spark, df are from the previous example
    # Print the schema in a tree format
    df.printSchema()
    # root
    # |-- age: long (nullable = true)
    # |-- name: string (nullable = true)


    # Select only the "name" column
    df.select("name").show()
    # +-------+
    # |   name|
    # +-------+
    # |Michael|
    # |   Andy|
    # | Justin|
    # +-------+


    # Select everybody, but increment the age by 1
    df.select(df['name'], df['age'] + 1).show()
    # +-------+---------+
    # |   name|(age + 1)|
    # +-------+---------+
    # |Michael|     null|
    # |   Andy|       31|
    # | Justin|       20|
    # +-------+---------+


    # Select people older than 21
    df.filter(df['age'] > 21).show()
    # +---+----+
    # |age|name|
    # +---+----+
    # | 30|Andy|
    # +---+----+


    # Count people by age
    df.groupBy("age").count().show()
    # +----+-----+
    # | age|count|
    # +----+-----+
    # |  19|    1|
    # |null|    1|
    # |  30|    1|
    # +----+-----+
    # $example off:untyped_ops$

    sqlDF = spark.sql("SELECT * FROM people")
    sqlDF.show()
    # +---+-------+
    # | age|   name|
    # +---+-------+
    # |null|Michael|
    # |  30|   Andy|
    # |  19| Justin|
    # +---+-------+
    # $example off:run_sql$


    # $example on:global_temp_view$
    # Register the DataFrame as a global temporary view
    df.createGlobalTempView("people")


    # Global temporary view is tied to a system preserved database `global_temp`
    spark.sql("SELECT * FROM global_temp.people").show()
    # +---+-------+
    # | age|   name|
    # +---+-------+
    # |null|Michael|
    # |  30|   Andy|
    # |  19| Justin|
    # +---+-------+
```

Dataframes Example

```
def basic_df_example(spark):
    # $example on:create_df$
    # spark is an existing SparkSession
    df = spark.read.json("examples/src/main/resources/people.json")
    # Displays the content of the DataFrame to stdout
    df.show()
    # +----+-------+
    # | age|   name|
    # +----+-------+
    # |null|Michael|
    # |  30|   Andy|
    # |  19| Justin|
    # +----+-------+
    # $example off:create_df$


    # $example on:untyped_ops$
    # spark, df are from the previous example
    # Print the schema in a tree format
    df.printSchema()
    # root
    # |-- age: long (nullable = true)
    # |-- name: string (nullable = true)


    # Select only the "name" column
    df.select("name").show()
    # +-------+
    # |   name|
    # +-------+
    # |Michael|
    # |   Andy|
    # | Justin|
    # +-------+


    # Select everybody, but increment the age by 1
    df.select(df['name'], df['age'] + 1).show()
    # +-------+---------+
    # |   name|(age + 1)|
    # +-------+---------+
    # |Michael|     null|
    # |   Andy|       31|
    # | Justin|       20|
    # +-------+---------+


    # Select people older than 21
    df.filter(df['age'] > 21).show()
    # +---+----+
    # |age|name|
    # +---+----+
    # | 30|Andy|
    # +---+----+


    # Count people by age
    df.groupBy("age").count().show()
    # +----+-----+
    # | age|count|
    # +----+-----+
    # |  19|    1|
    # |null|    1|
    # |  30|    1|
    # +----+-----+
    # $example off:untyped_ops$

    sqlDF = spark.sql("SELECT * FROM people")
    sqlDF.show()
    # +---+-------+
    # | age|   name|
    # +---+-------+
    # |null|Michael|
    # |  30|   Andy|
    # |  19| Justin|
    # +---+-------+
    # $example off:run_sql$


    # $example on:global_temp_view$
    # Register the DataFrame as a global temporary view
    df.createGlobalTempView("people")


    # Global temporary view is tied to a system preserved database `global_temp`
    spark.sql("SELECT * FROM global_temp.people").show()
    # +---+-------+
    # | age|   name|
    # +---+-------+
    # |null|Michael|
    # |  30|   Andy|
    # |  19| Justin|
    # +---+-------+
```

Summary

- Spark is a popular and widely used framework for large-scale computing
- Simple programming interface
  - You don't need to typically worry about the parallelization
  - That's handled by Spark transparently
  - In practice, may need to fiddle with number of partitions etc.
- Managed services supported by several vendors including Databricks (started by the authors of Spark), Cloudera, etc.
- Many other concepts that we did not discuss
  - Shared accumulator and broadcast variables
  - Support for Machine Learning, Graph Analytics, Streaming, and other use cases
- Alternatives include: Apache Tez, Flink, and several others

Outline

- Data Models: Then, and now
  - History of Data Models (“what comes around…”)
  - A data model for Key-value Stores (“a co-relational model..”)
- Languages
  - Overview
  - Datalog (“a survey of research…” and “declarative networking…”)
- Map-reduce and Spark
  - Original MR Abstraction (”mapreduce:” …”)
  - Spark (“resilient distributed datasets…”)
- SystemML: An abstraction for ML
- GraphX: An abstraction for Graphs

Motivation

- MapReduce becoming a popular platform for distributed processing, but not at the right level of abstraction for ML
- ML programs need to be written at the level of matrices, vectors, and tensors
  - Think R, or Python Libraries like Scikit (and PyTorch today)
- Many core ML operations on matrices are “summations” and can be done as “group by aggregates”
- Need a high-level abstraction that allows arbitrary ML programs
  - Then we can optimize how it maps to Hadoop MapReduce (or Spark)

Simpler Example

- From: MapReduce for Machine Learning on Multicore; 2005
- Linear Regression
  - Given training data (**x1**, y1), (**x2**, y2), …, fit a model: $y = \theta^T x$ that minimizes loss, where **x1**, **x2**, .., are vectors

x_{11}	x_{12}	…	…	…	…	x_{1m}
x_{21}	x_{22}	…	…	…	…	x_{2m}
…	…	…	…	…	…	…
…	…	…	…	…	…	…
x_{N1}	x_{N2}	…	…	…	…	x_{Nm}

X

y_1
y_2
…
…
y_N

Y

N >>> m

  - Least square estimate: $\theta^* = A^{-1}b$, where:
    - A is a m-by-m matrix = X^TX, and b is a m-by-1 matrix = X^TY

Simpler Example: Computing A

Exploits the fact that: $X^TX = SIGMA\ (x_i^Tx_i)$

Each mapper computes: $x_i^Tx_i$ for a group of rows in X

The output of each mapper is: m-by-m matrix

A single reducer sums up all of those matrixes to get the final m-by-m matrix

Similarly we can compute: $b = SIGMA\ (x_i^T y_i)$

x_{11}	x_{12}	…	…	…	…	x_{1m}
x_{21}	x_{22}	…	…	…	…	x_{2m}
…	…	…	…	…	…	…
…	…	…	…	…	…	…
x_{N1}	x_{N2}	…	…	…	…	x_{Nm}

X

Simpler Example: Computing A

Exploits the fact that: $X^TX = SIGMA (x_i^Tx_i)$

Each mapper computes: $x_i^Tx_i$ for a group of rows in X

The output of each mapper is: m-by-m matrix

A single reducer sums up all of those matrixes to get the final m-by-m matrix

Similarly we can compute: $b = SIGMA (x_i^T y_i)$

Only works if m <<< N

- So the communication is m-by-m matrices or m-by-1 vectors
- Each mapper deals with much more data

Example Program

Algorithm 1 Gaussian Non-Negative Matrix Factorization

1: V = read("in/V"); //read input matrix V
2: W = read("in/W"); //read initial values of W
3: H = read("in/H"); //read initial values of H
4: max_iteration $= 20$;
5: $i = 0$;
6: **while** i $<$ max_iteration **do**
7: $H = H * (W^T V \ / \ W^T W H)$; //update H
8: $W = W * (V H^T \ / \ W H H^T)$; //update W
9: $i = i + 1$;
10: **end while**
11: write(W,"out/W"); //write result W
12: write(H,"out/H"); //write result H

Example: Two of the Options

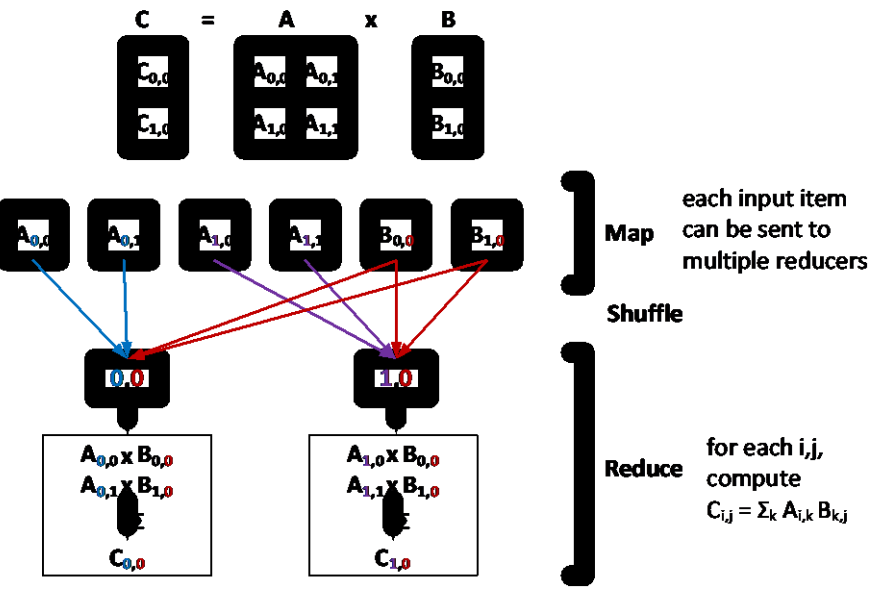


Fig. 1. RMM: Replication based Matrix Multiplication

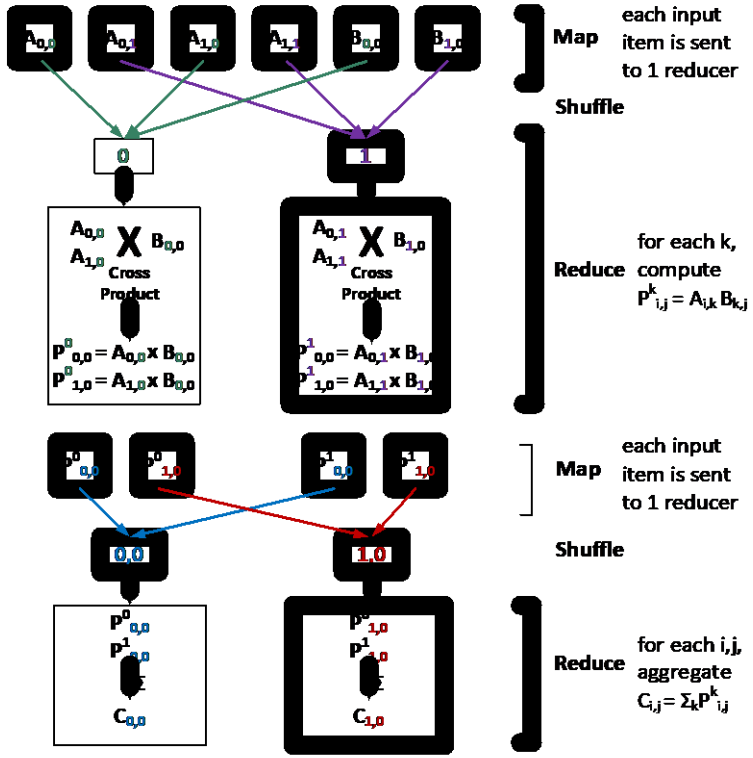
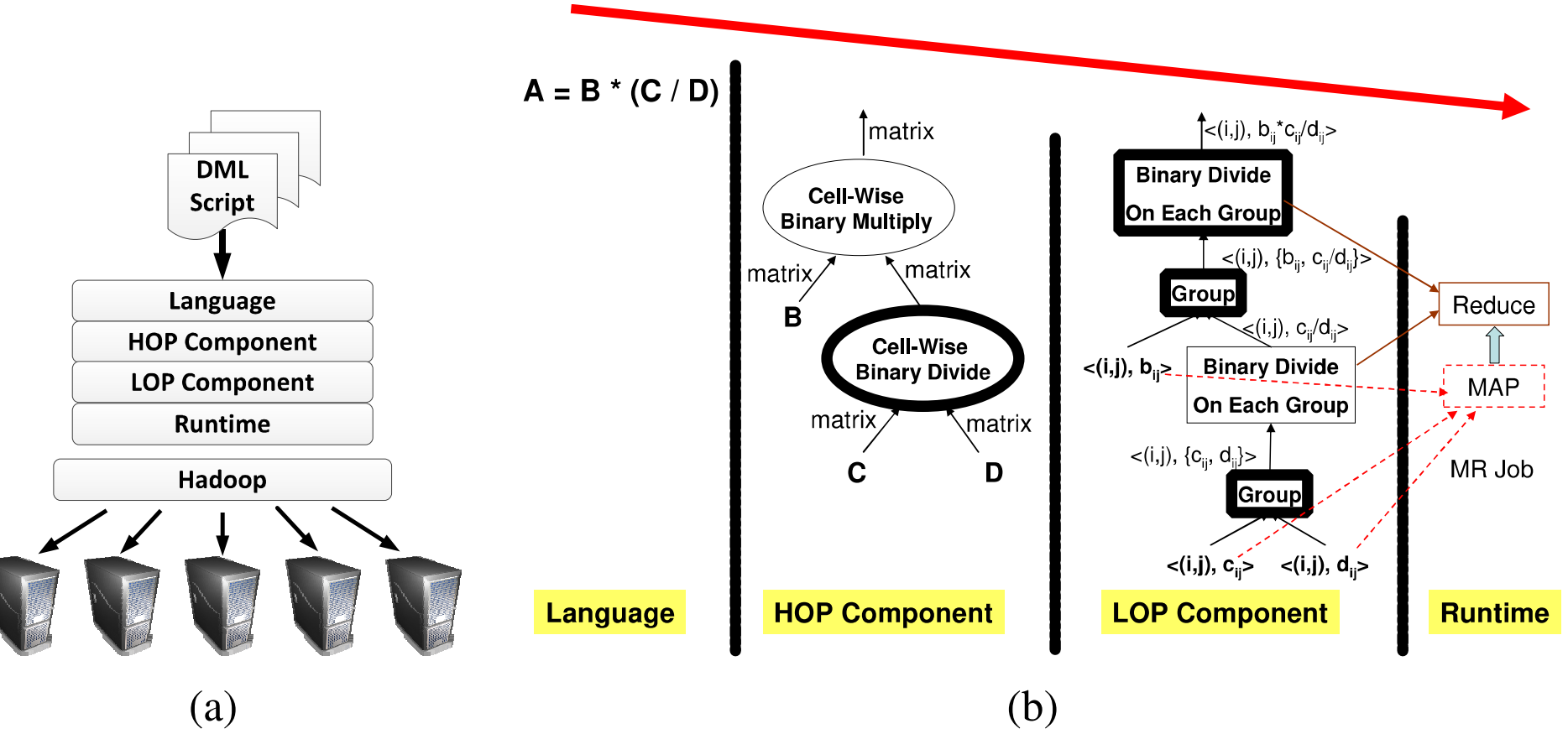


Fig. 2. CPMM: Cross Product based Matrix Multiplication

SystemML Architecture



DML Language

- Based on R
  - Somewhat restricted to be able to optimize
- Two Data Types: Matrices and Scalars

Algorithm 1	DML Statement	Semantics	HOP Notation	LOP Notation
$Z = X * Y$	`Z=X*Y`	cell-wise multiplication: $z_{ij} = x_{ij} * y_{ij}$	$b(*) : X, Y$	$group \rightarrow binary(*)$
$Z = X/Y$	`Z=X/Y`	cell-wise division: $z_{ij} = x_{ij}/y_{ij}$	$b(/) : X, Y$	$group \rightarrow binary(/)$
$Z = XY$	`Z=X%*%Y`	matrix multiplication: $z_{ij} = \sum_k x_{ik} * y_{kj}$	$ab(\sum, *) : X, Y$	$(mmrj)$ or $(mmcj \rightarrow group \rightarrow aggregate(\sum))$
$Z = X^T$	`Z=t(X)`	transpose: $z_{ij} = x_{ji}$	$r(T) : X$	$transform(t)$
	`Z=log(X)`	cell-wise logarithm: $z_{ij} = log(x_{ij})$	$u(log) : X$	$unary(log)$
	`Z=rowSum(X)`	row-wise sums: $z_i = \sum_j x_{ij}$	$au(\sum, row) : X$	$transform(row) \rightarrow group \rightarrow aggregate(\sum)$

Developments since then

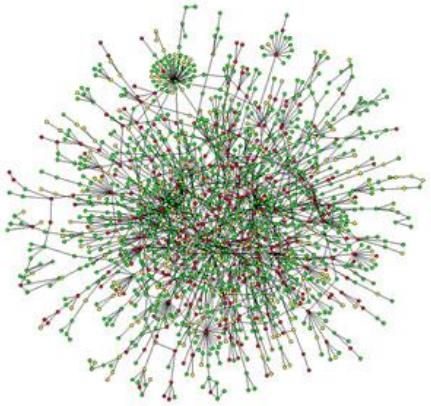
- Much followup work on linear algebra-based frameworks
- SciDB also has an array data model
  - But different types of operations
- Apache Spark has MLIB
  - More of a ”library” flavor
  - Many implementations of popular algorithms – some not easy to map to MR
- Development of deep learning frameworks like TensorFlow and PyTorch
- Also much other work on custom ML platforms (e.g., parameter server)

Outline

- Data Models: Then, and now
  - History of Data Models (“what comes around…”)
  - A data model for Key-value Stores (“a co-relational model..”)
- Languages
  - Overview
  - Datalog (“a survey of research…” and “declarative networking…”)
- Map-reduce and Spark
  - Original MR Abstraction (”mapreduce:” …”)
  - Spark (“resilient distributed datasets…”)
- SystemML: An abstraction for ML
- GraphX: An abstraction for Graphs

Motivation

- Lot of graph structure in real-world datasets



A protein-protein interaction network

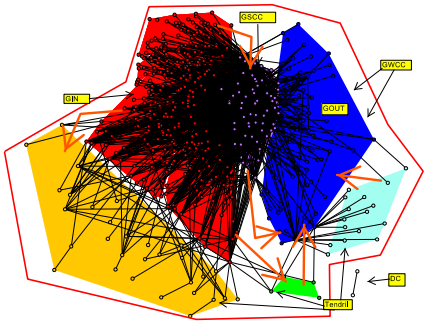
Citation networks

Communication networks

Disease transmission networks

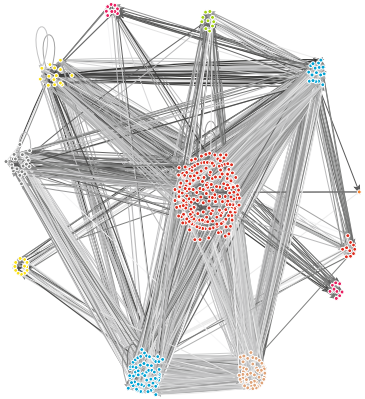
facebook twitter LinkedIn

Social networks

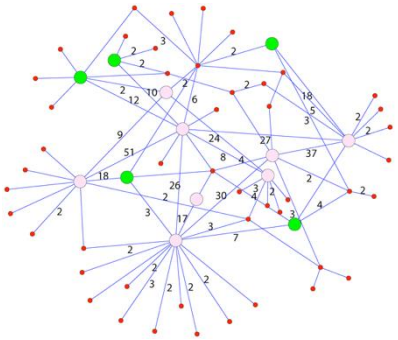


Federal funds networks

Knowledge Graph



Financial transaction networks



Stock Trading Networks

Motivation

- Lot of graph structure in real-world datasets
- Types of analytics of interest
  - Centrality analysis (find most central nodes), Community detection, Vulnerability analysis, …
- Different from "graph queries"
  - e.g., find shortest path between nodes, or find specific patterns in graph

- GraphX and most other systems like it (e.g., Google Pregel, Apache Giraph) fairly limited in what they can support (e.g., can only do basic centrality or community detection)
- More powerful frameworks in HPC community

Pregel

- Pregel uses a “vertex-centric programming model”
- In each iteration, receive messages from neighbors, do some computations, send messages to neighbors

```
def PageRank(v: Id, msgs: List[Double]) {
  // Compute the message sum
  var msgSum = 0
  for (m <- msgs) { msgSum += m }
  // Update the PageRank
  PR(v) = 0.15 + 0.85 * msgSum
  // Broadcast messages with new PR
  for (j <- OutNbrs(v)) {
    msg = PR(v) / NumLinks(v)
    send_msg(to=j, msg)
  }
  // Check for termination
  if (converged(PR(v))) voteToHalt(v)
}
```

Listing 1: **PageRank in Pregel**: computes the sum of the inbound messages, updates the PageRank value for the vertex, and then sends the new weighted PageRank value to neighboring vertices. Finally, if the PageRank did not change the vertex program votes to halt.

GraphX

- A graph represented by:
  - Vertex collection (RDD): a collection of vertices, with their IDs and properties
  - Edge collection (RDD): keyed by the pair of vertex IDs, and stores the properties
- Can manipulate the above two collections using Spark operations
  - e.g., add new properties by joining with another collection, etc.
- Triplets View: Join the two collections to append the source/target properties with each edge

GraphX

Key operation:

- does a local computation for each edge,
- followed by a group by aggregate on Vertex ID



```
class Graph[V, E] {
  // Constructor
  def Graph(v: Collection[(Id, V)],
            e: Collection[(Id, Id, E)])
  // Collection views
  def vertices: Collection[(Id, V)]
  def edges: Collection[(Id, Id, E)]
  def triplets: Collection[Triplet]
  // Graph-parallel computation
  def mrTriplets(f: (Triplet) => M,
      sum: (M, M) => M): Collection[(Id, M)]
  // Convenience functions
  def mapV(f: (Id, V) => V): Graph[V, E]
  def mapE(f: (Id, Id, E) => E): Graph[V, E]
  def leftJoinV(v: Collection[(Id, V)],
      f: (Id, V, V) => V): Graph[V, E]
  def leftJoinE(e: Collection[(Id, Id, E)],
      f: (Id, Id, E, E) => E): Graph[V, E]
  def subgraph(vPred: (Id, V) => Boolean,
      ePred: (Triplet) => Boolean)
    : Graph[V, E]
  def reverse: Graph[V, E]
}
```

Listing 4: Graph Operators: transform vertex and edge collections.

GraphX: Example

```
val graph: Graph[User, Double]
def mapUDF(t: Triplet[User, Double]) =
  if (t.src.age > t.dst.age) 1 else 0
def reduceUDF(a: Int, b: Int): Int = a + b
val seniors: Collection[(Id, Int)] =
  graph.mrTriplets(mapUDF, reduceUDF)
```

Figure 2: Example use of mrTriplets: Compute the number of older followers of each vertex.

GraphX: Pregel Implementation

```
def Pregel(g: Graph[V, E],
       vprog: (Id, V, M) => V,
       sendMsg: (Triplet) => M,
       gather: (M, M) => M): Collection[V] = {
 // Set all vertices as active
 g = g.mapV((id, v) => (v, halt=false))
 // Loop until convergence
 while (g.vertices.exists(v => !v.halt)) {
   // Compute the messages
   val msgs: Collection[(Id, M)] =
     // Restrict to edges with active source
     g.subgraph(ePred=(s, d, sP, eP, dP) =>!sP.halt)
     // Compute messages
      .mrTriplets(sendMsg, gather)
   // Receive messages and run vertex program
   g = g.leftJoinV(msgs).mapV(vprog)
 }
 return g.vertices
}
```

Listing 5: GraphX Enhanced Pregel: An implementation of the Pregel abstraction using the GraphX API.

Developments since then

- Lot of work on more efficient implementation of Pregel-like systems
- Other more expressive programming frameworks
  - Arabesque: for graph mining
  - NScale (by us here): generalization to allow computations on multi-hops
  - Green-marl, Ligra and successors: lower-level programming frameworks designed for HPC environments
- Unclear how widely any of these are used in practice