

Introduction to Database Systems

I2DBS – Spring 2023

- Week 1:
- Lecture Introduction
- DBMS Introduction
- Relational Model
- SQL

Jorge-Arnulfo Quiané-Ruiz

Readings:

PDBM 1, 6.1, 7.1-7.2

Course Responsible

Jorge Arnulfo Quiané Ruiz

- **Associate Professor** 01/2023 -- present
IT University of Copenhagen
- **Principal Researcher** 11/2019 – 12/2022
TU Berlin
- **Ph.D. in Computer Science** 09/2008
Inria & University of Nantes
- **Website:** <https://www.user.tu-berlin.de/quiane/>



Cool TAs



Mille Mei Zhen Loo (Mille)

Study program: SWU -- 6th semester
Preferred hobby: Knitting



Nikoline Burman (Niko)

Study program: SWU -- 6th semester
Preferred hobby: Board games



Deniz Yildirim (Deniz)

Study program: SWU -- 6th semester
Preferred hobby: League of legends



Frederik Rothe (Freddy)

Study program: SWU -- 6th semester
Preferred hobby: Endurance sports



Frederik Hongaard (Fred)

Study program: MSc. CS – 2nd semester
Preferred hobby: Rowing



Emil Ravn (Emil)

Study program: CS – 2nd semester
Preferred hobby: learning different food techniques

Research Group

<https://dasya.itu.dk>

20+ People

IT University of Copenhagen

Lab for research and Education: 5th floor (5A56)

Entire Data Lifecycle:

- Collection
- Transfer
- Storage
- Curation
- Processing
- Analytics

Infrastructure for Data Science

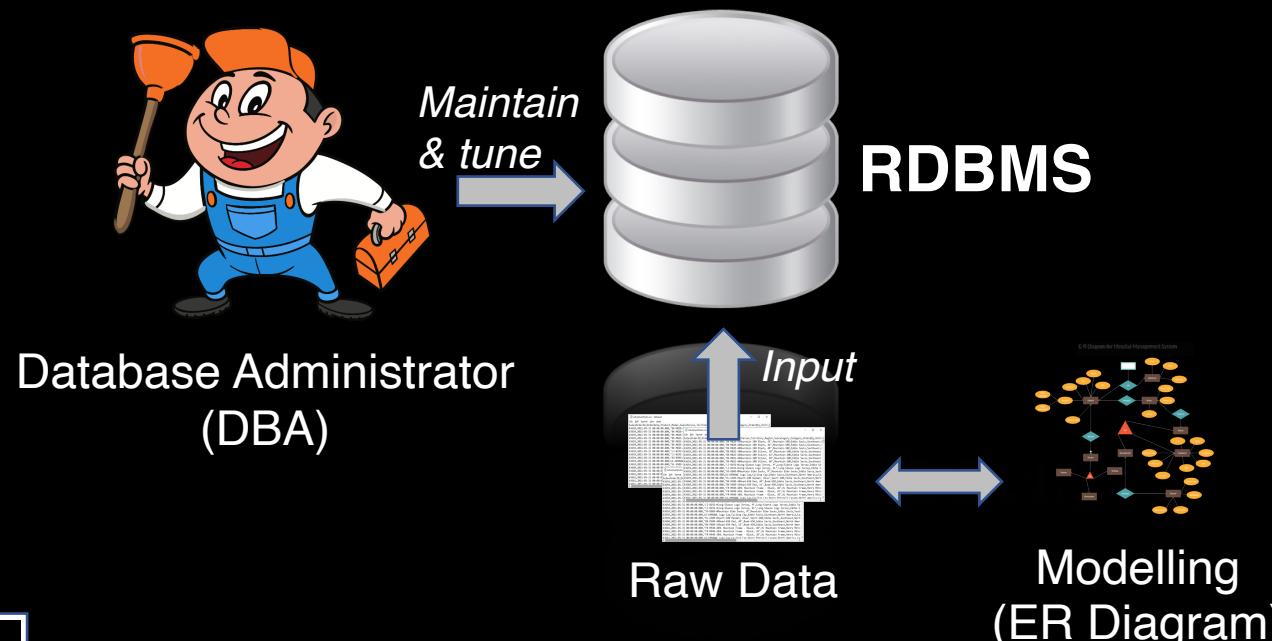
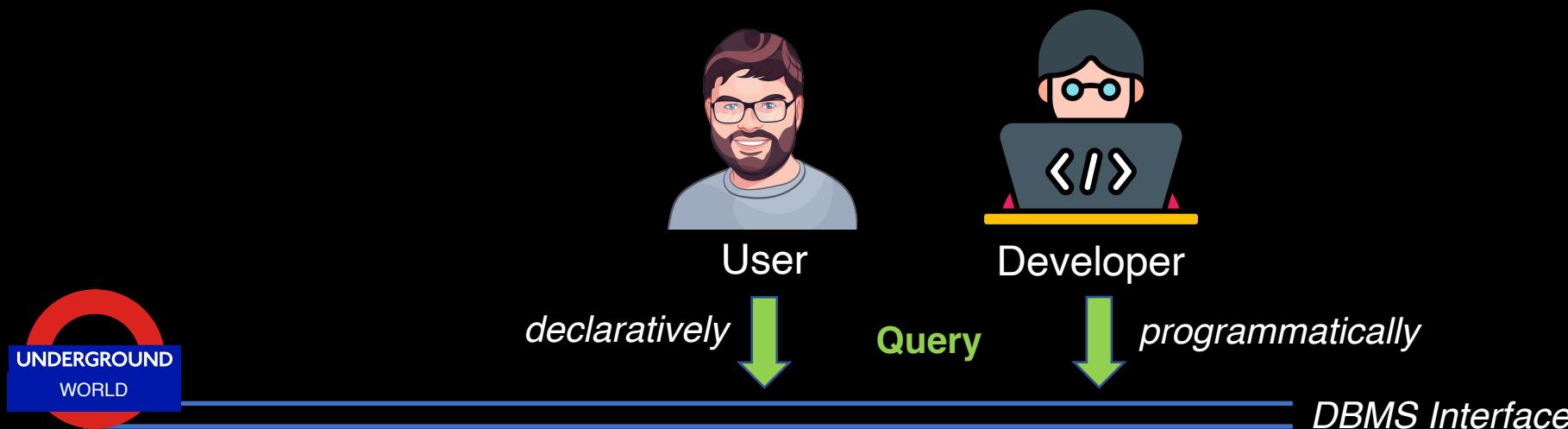


Data-Intensive Systems and Applications

But This is About You!

- ~150 Students
Mostly from MSc. in Software Design
- Too many for a round table
Let's run a round table based on stratified sampling
- What do you expect from I2DBS?
Let's ask randomly

What You will Learn



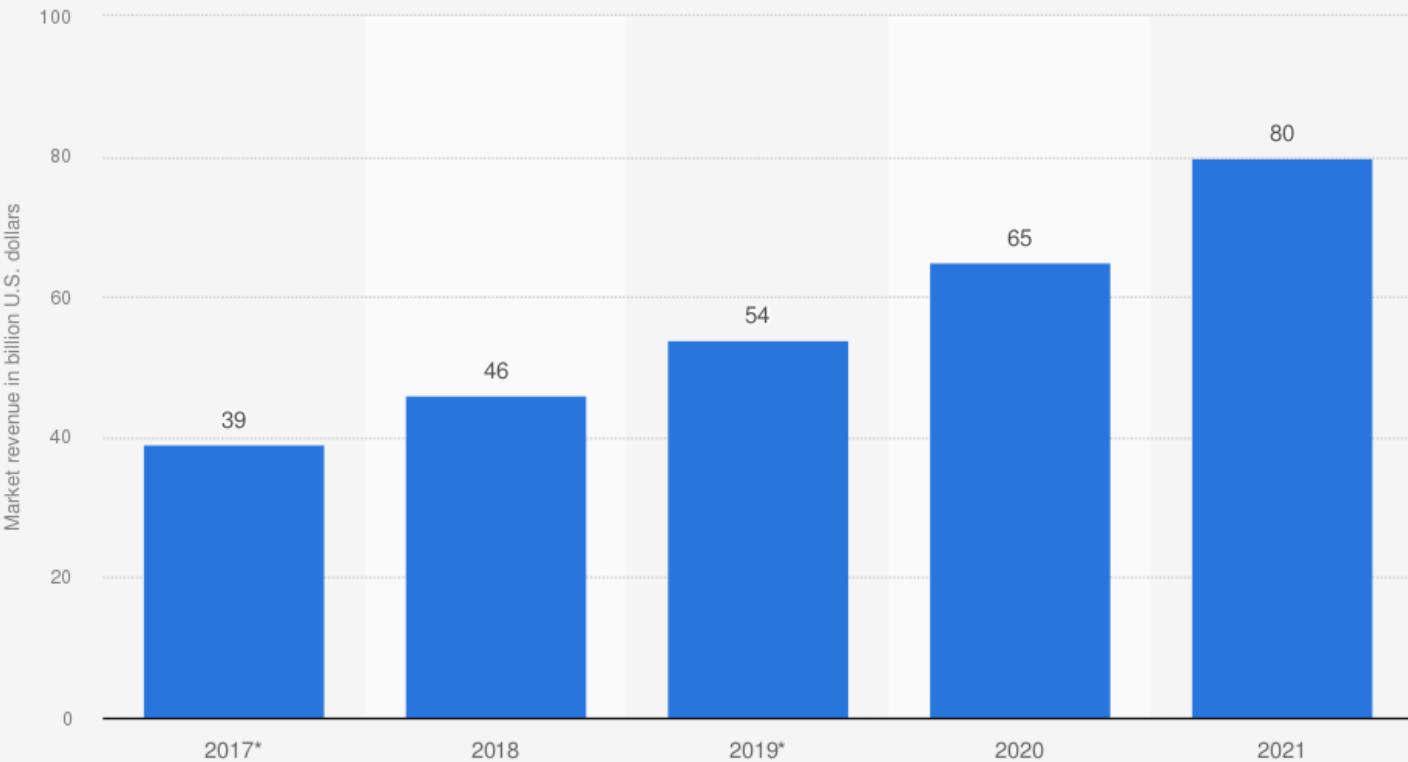
Intended Learning Outcomes

- Write SQL queries: multiple relations; compound conditions; grouping; aggregation; and subqueries.
 - Use relations
 - Suggest a design in a suitable manner.
 - Analyze/preprocess data in a secure manner.
 - Reflect upon the evolution of the hardware and storage hierarchy and its impact on data management system design.
 - Discuss the pros and cons of different classes of data systems for modern analytics and data science applications.
1. Getting into Database Systems
 2. Getting data using SQL and from your apps
 3. Design a database
 4. Tune a database
 5. Advanced databases (internals and big data)

Why is It Important?

- Crucial to effectively manage and utilize data
- Help to maintain data integrity and security
- Ease app development

Size of the database management system (DBMS) market worldwide from 2017 to 2021 (in billion U.S. dollars)



Sources
Gartner; Statista
© Statista 2022

Additional Information:
Worldwide; Gartner; Statista; 2017 to 2021

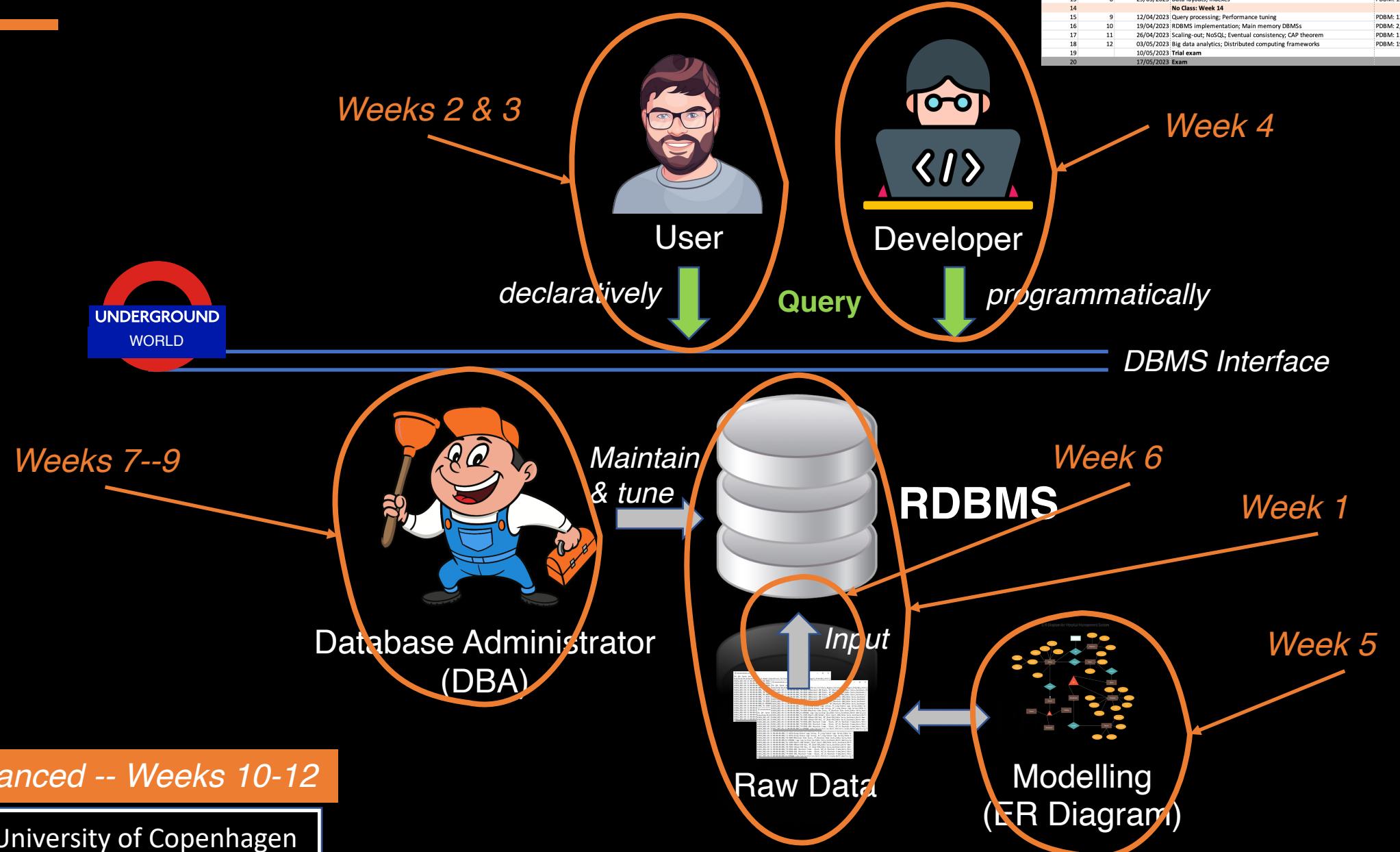
Course Schedule

• BSc vs MSc

- *Pretty much the same course*
- *One different question (5%) on exam*

Week	Lecture	Date	Topic	Readings	Exersice/Homework	Lecturer
5	1	01/02/2023	Introduction; Relational data model; SQL DDL/DML	PDBM: 1, 6.1, 7.1-7.2	Exercise 1	Jorge Quiané
6	2	08/02/2023	Basic SQL queries; Joins; Aggregations	PDBM: 7.3	Exercise 2	Jorge Quiané
7	3	15/02/2023	Complex SQL queries; Subqueries; Views	PDBM: 7.3-7.4	Homework 1 (due on 22.02.2023)	Jorge Quiané
8	4	22/02/2023	[BSc] Triggers; Functions; Transactions; Using SQL from Java (basics)	PDBM: 9.2, 14.1, 14.2.1, 14.5 + more	Exercise 4B	Jorge Quiané
			[MSc] DBMS programming in Java	Details below		Jorge Quiané
9		01/03/2023	No Lecture (video lecture for graduate students)		Exercise 4M	
10	5	08/03/2023	ER diagrams; Translation to SQL databases	PDBM: 3.0-3.3,6.3-6.4	Exercise 5	Jorge Quiané
11	6	15/03/2023	Normalization	PDBM: 6.2-6.4	Homework 2 (due on 22.03.2023)	Jorge Quiané
12	7	22/03/2023	Storage hierarchy; Multicore processing; OS	Videos & Chapter (details below)	Exercise 7	Pinar Tözün
13	8	29/03/2023	Data layouts; Indexes	PDBM: 12	Exercise 8	Jorge Quiané
14			No Class: Week 14			
15	9	12/04/2023	Query processing; Performance tuning	PDBM: 13.1	Homework 3 (due on 19.04.2023)	Jorge Quiané
16	10	19/04/2023	RDBMS implementation; Main memory DBMSs	PDBM: 2, 14 + more (details below)	Old exercises/exams	Jorge Quiané
17	11	26/04/2023	Scaling-out; NoSQL; Eventual consistency; CAP theorem	PDBM: 11 + (optional papers)	Old exercises/exams	Jorge Quiané
18	12	03/05/2023	Big data analytics; Distributed computing frameworks	PDBM: 19.1-19.2, 19.4, 20.1-20.3	Homework 4 (due on 10.05.2023)	Jorge Quiané
19		10/05/2023	Trial exam			Jorge Quiané
20		17/05/2023	Exam			

Course Schedule (Illustrated)



Course Structure

- **Lectures**
 - *Wednesdays 12.15 – 14:00 at Aud0 (0A27)*
 - *Preparation required (read the material)*
- **Exercises**
 - *Wednesdays 14.15 – 16:00 at 2A12-14, 3A12-14, and 4A58.*
 - *Preparation not required (related to previous lectures)*
- **Homeworks**
 - *4 homeworks (deadlines to be published on learnIT)*
 - *Mandatory, yet easy to get accepted!*
 - *Feedback in the following weeks (if submitted on time)*
- **learnIT:** course outline, materials, announcements, ...
- **Piazza:** Q&A, messages, and updates.
- **Office hours:** private message to Jorge on Piazza (asking your question)

Note

Exercise1 Rooms per OS:

- Windows (3A12-14)
- Mac (2A12-14)
- Linux (4A58)

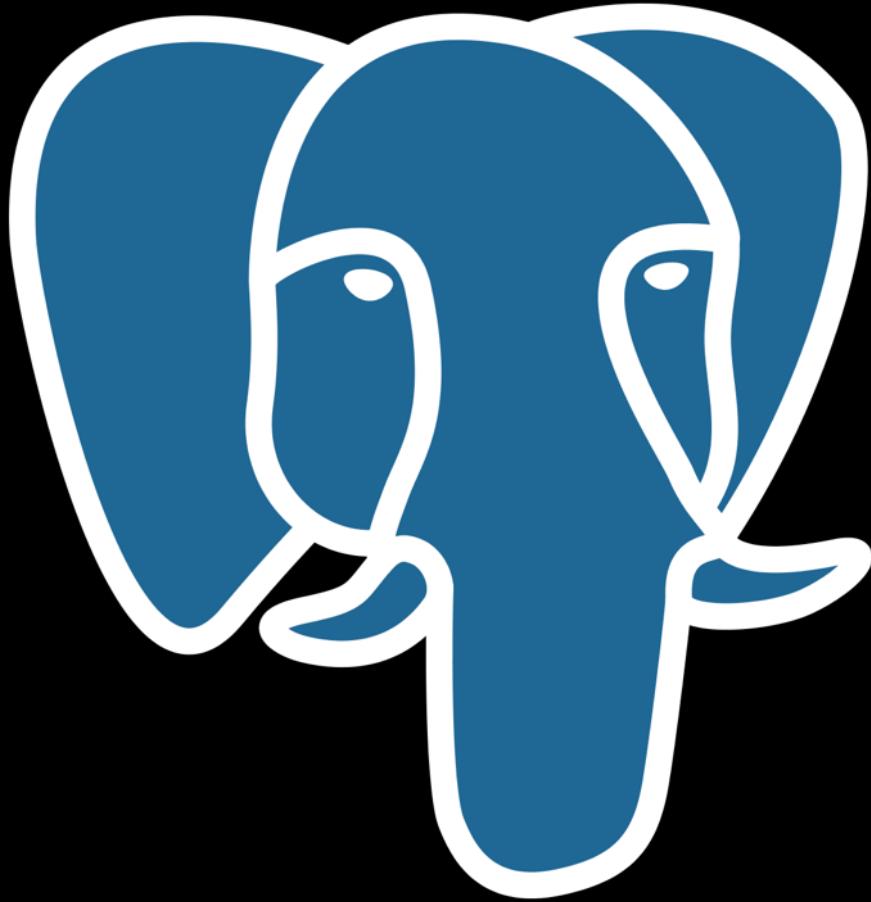
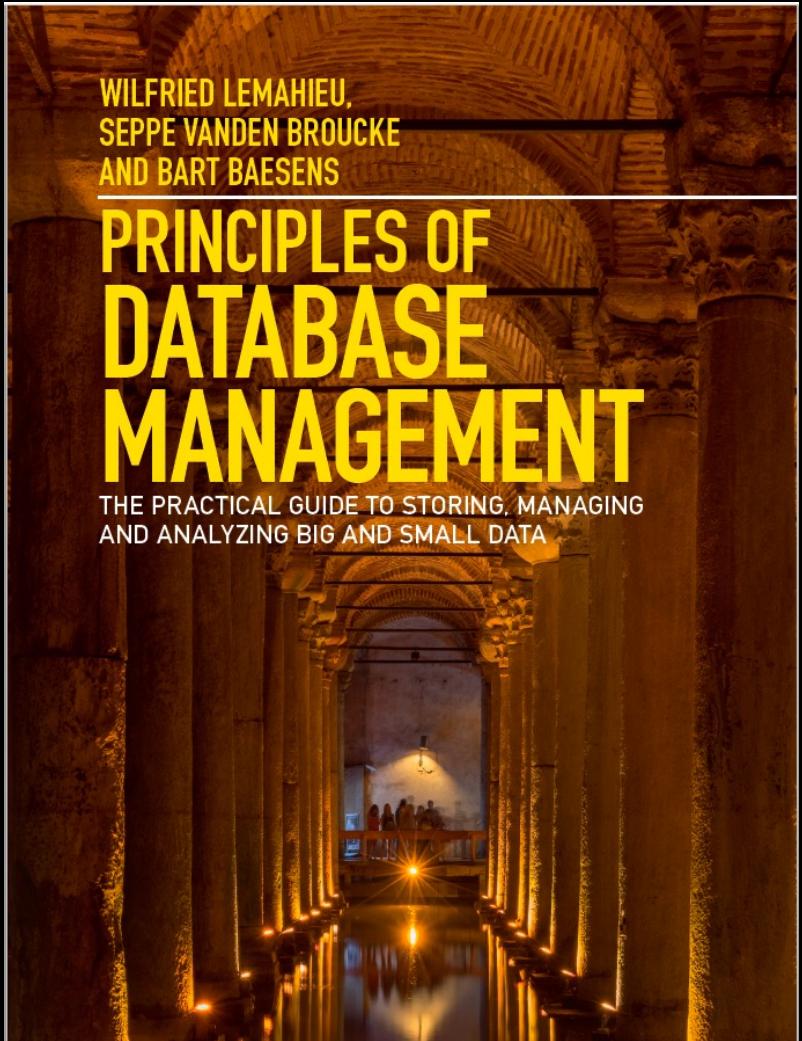
Course Methodology

- You learn: We are here to help you!
- You need to read the book beforehand
 - Yes, we often assume you have done so
 - All readings are in the schedule on learnIT
- We work in a pull model fashion: ask questions!

Advise

Prepare and Ask questions

Book and Database System



PostgreSQL
(pgAdmin + psql)

How will We Assess your Learnings?

- **100% Exam (Quiz on learnIT)**
 - All course materials are allowed
 - Communication is not allowed though
- **Exercises and homeworks will also prepare you!**

Advise

Study the material weekly

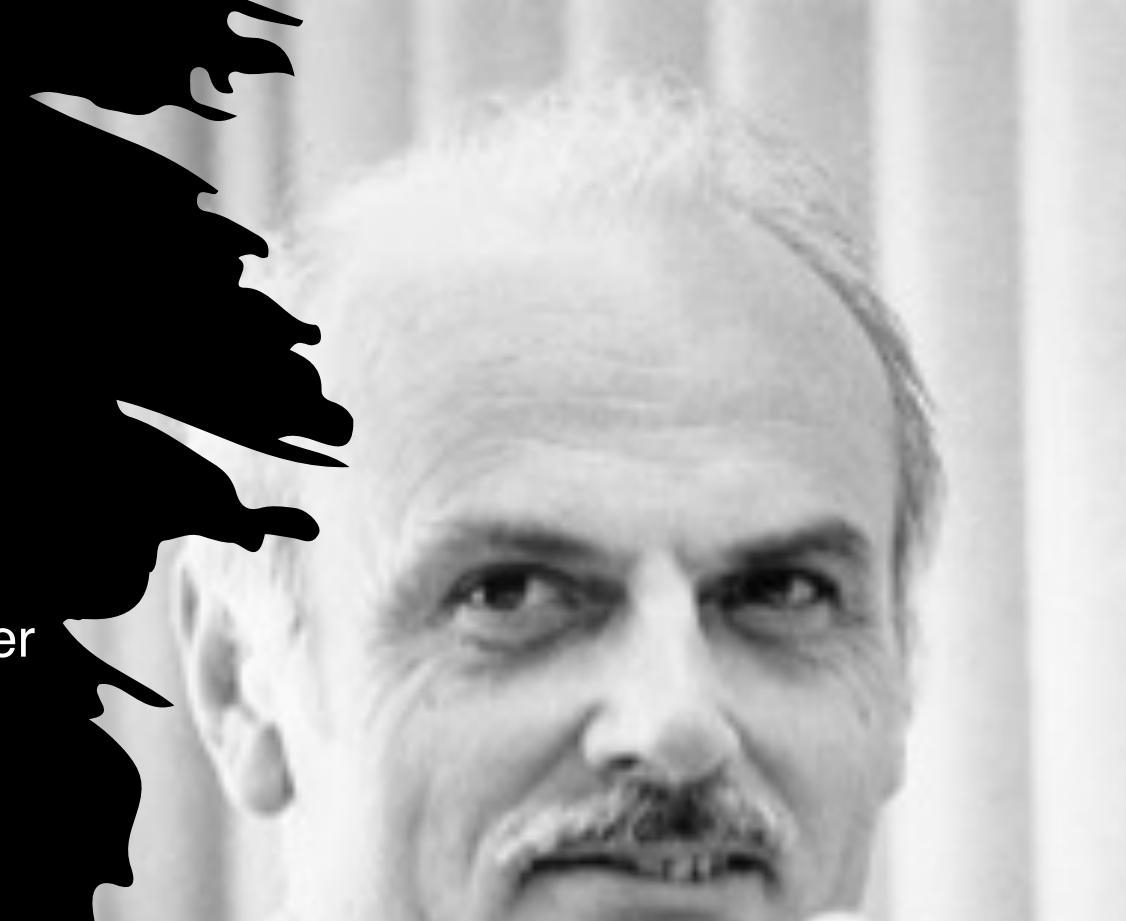


Profile of the Week

Edgar F. Codd

Father of Databases (Relational Model)

- 1923: Born 23/8, Isle of Portland, England
- 1965: PhD in CS from University of Michigan
- 1967: Moved to IBM Almaden Research Center
- 1969: Invented the relational model
- 1976: IBM Fellow
- 1981: Turing Award
- 1994: ACM Fellow



Information Retrieval

P. BAXENDALE, Editor

A Relational Model of Data for Large Shared Data Banks

E. F. CODD

IBM Research Laboratory, San Jose, California

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] presently in vogue for non-inferential systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representa-



RDBMS

DBMS Brief Introduction

Readings:

PDBM 1

Database Definition

- A **database** is a collection of related data items within a specific business process or problem setting
- A **database system** provides a way to systematically organize, store, retrieve, and query a database

Cons

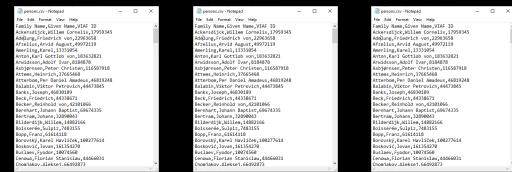
- Redundant data
- Inconsistent data
- **Data dependency**
- Limited concurrency
- Expensive maintenance
- Poor performance

File-based Database

Apps



File storage



Relational Database

- A **relational database** is a type of database that is based on the **relational model**
 - stores data in a set of tables with rows and columns (a.k.a. relations)
 - uses relationships between these tables to manage the data.
- A **relational database system** (RDBMS) implements and manages relational databases.

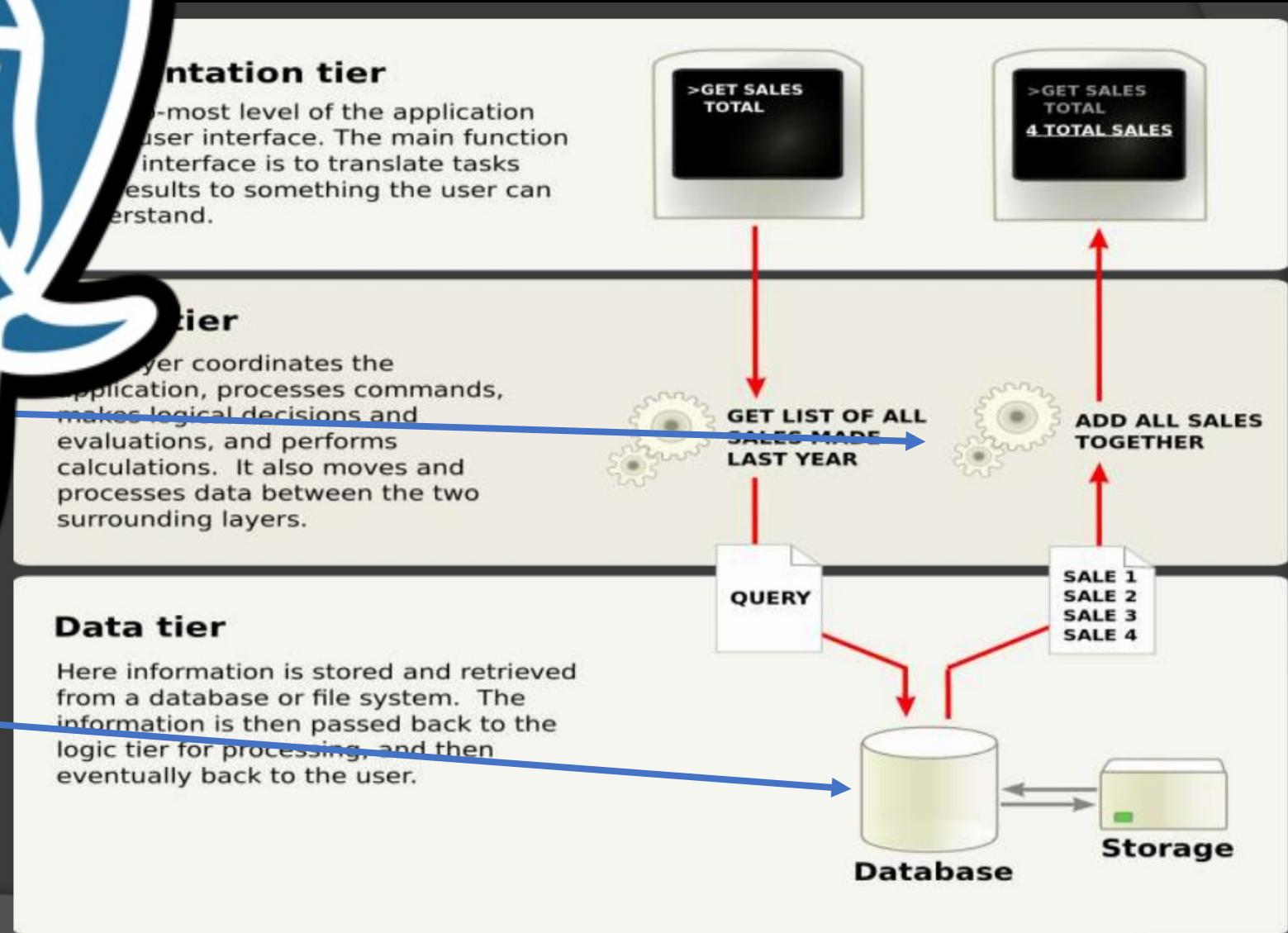
● **Pros**

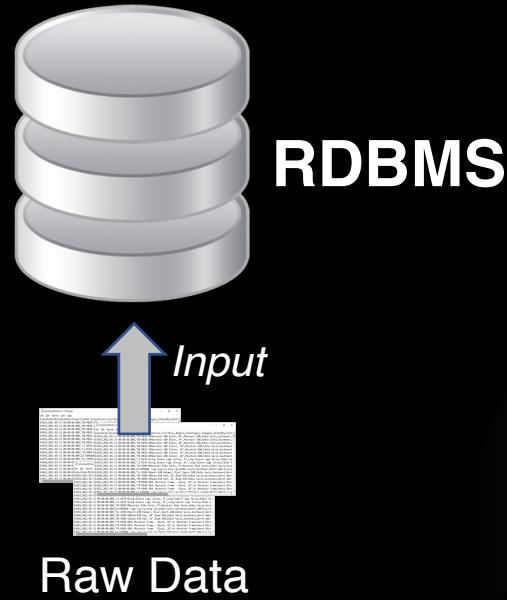
- Unique data entities
- Data integrity
- **Data independency**
- High concurrency
- Cheap maintenance
- High performance

Applications

py. JAR

PostgreSQL





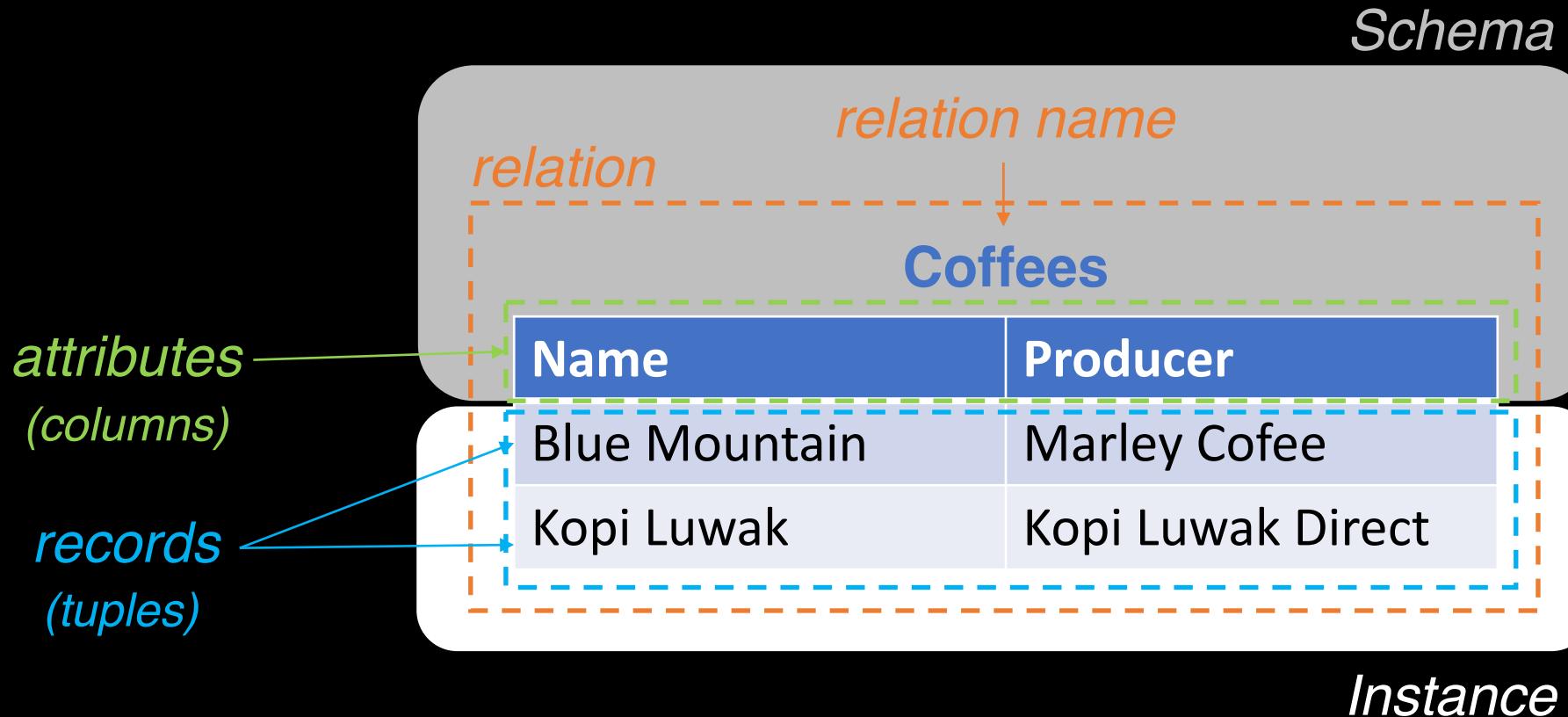
Relational Model

Readings:

PDBM 1, 6.1

Basic Concepts...

Relation



Basic Concepts...

Schema vs Instance vs Database

- **Relation**

- **Schema = name + list of attributes**
 - *Optional: attributed types*
 - *Coffees (Name, Producer)*
 - *Coffees (Name:STRING, Producer:String)* -- specifies the domain
- **Instance**
 - *Records in a relation*
 - *E.g., (Blue Mountain, Marley Coffee)*

- **Database = collection of relations**

- *Database schema = set of all relations names in the database*
- *Database instance = set of all relations instances in the database*

Basic Concepts...

Example of a Database Schema

- **Students** (SId:INT, Name:STRING, Email:STRING, Semester:INT)
- **Faculty** (FId:INT, Name:STRING, DId:INT)
- **Courses** (CId:STRING, Name:STRING, DId:INT)
- **Departments** (DId:INT, Name:STRING)
- **Transcripts** (CId:STRING, SId:INT, Grade:GRADES, Comment:STRING)

Why Relations?

- **Pros**

- Very simple model
- How we typically think about structured data
- Conceptual model behind SQL, which is the most important query language today

- **Cons**

- Too simple for some things: hierarchical multi-valued data
- Much of today's data is NOT structured
- It could be complex to implement well

Identifiers

identifier

- **Students** (SId:INT, Name:STRING, Email:STRING, Semester:INT)
- **Faculty** (FId:INT, Name:STRING, DId:INT)
- **Courses** (CId:STRING, Name:STRING, DId:INT)
- **Departments** (DId:INT, Name:STRING)
- **Transcripts** (CId:STRING, SId:INT, Grade:GRADES, Comment:STRING)

Keys in Relations

Keys and Superkeys

- **What is a key?**
 - Defines unique records (instances)
 - Helps in setting relationships between relations
 - Ensures the mathematical definition of a relation (*set* of records)
- **Superkeys**
 - Is a set of attributes that uniquely identify records: *Uniqueness* property
 - The entire set of attributes of a relation is a superkey
 - Minimal superkey: *Minimality* property
 - No attribute can be removed from a superkey without violating the uniqueness property

Students

SId	Name	Email	Semester
01785	Bob Brown	bobr@itu.dk	2
01615	Lucas White	luwh@itu.dk	5

A yellow oval highlights the first three columns (SId, Name, Email). A red circle highlights the SId column. A yellow arrow labeled "superkey" points to the yellow oval. A red arrow labeled "key" points to the red circle.

Keys in Relations

Candidate Keys

- Attributes that satisfies the uniqueness and minimality properties
 - Minimal superkey = (candidate) key
 - Superkeys contains at least one (candidate) key
 - A relation can have many (candidate) keys

The diagram shows a relation named "Students" with four columns: SId, Name, Email, and Semester. The first three columns are highlighted with a yellow oval, and the entire oval is circled in red. A red 'X' is placed above the oval with the label "superkey". A green checkmark is placed above the "Email" column with the label "key".

SId	Name	Email	Semester
01785	Bob Brown	bobr@itu.dk	2
01615	Lucas White	luwh@itu.dk	5

Keys in Relations

Primary Keys

- A key to identify records in a relation
 - Important to define indexes and for storage purposes (later in the course)
 - Cannot be NULL
 - Also used to establish relationships with other relations
 - From all candidate keys only one can be primary key
 - The remaining ones are known as *Alternative Keys*

The diagram shows a table titled "Students" with columns: SId, Name, Email, and Semester. The "SId" column is highlighted with a red oval and labeled "superkey". The "Email" column is highlighted with a yellow oval and labeled "key". A green checkmark is placed above the "Email" column header.

SId	Name	Email	Semester
01785	Bob Brown	bobr@itu.dk	2
01615	Lucas White	luwh@itu.dk	5

Keys in Relations

Live Exercise Students

- **What are superkeys and keys?**

- (SId) -- (Candidate) Key
- (Email) -- (Candidate) Key
- (SId, Name) -- Superkey
- (Semester) -- none
- (Email, Semester) -- Superkey
- (Name) -- none
- (Name, Semester) -- (Candidate) Key

SId	Name	Email	Semester
01785	Bob Brown	bobr@itu.dk	2
01615	Lucas White	luwh@itu.dk	5
01436	Olga Marx	olma@itu.dk	6
01875	Jens Schuh	jesc@itu.dk	1
01803	Olga Marx	olmr@itu.dk	2
01567	Peter Pitt	pepi@itu.dk	1

- **What is the best key for being the primary key?**

(SId) is ideal for being the Primary Key

- **Which of these keys (does) not make sense in practice?**

(Name, Semester)

Relationships

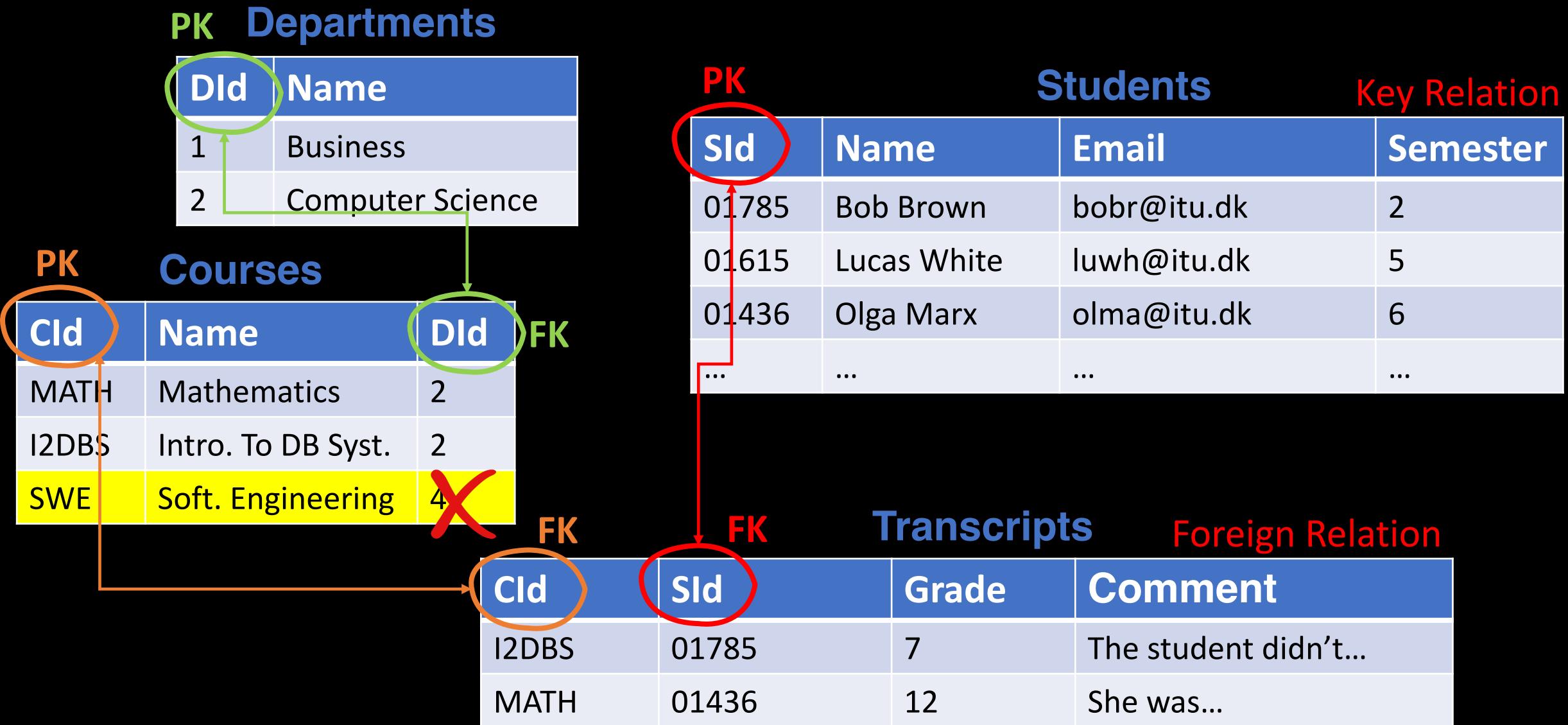
- **Students** (SId:INT, Name:STRING, Email:STRING, Semester:INT)
- **Faculty** (FId:INT, Name:STRING, DId:INT)
- **Courses** (CId:STRING, Name:STRING, DId:INT)
- **Departments** (DId:INT, Name:STRING)
- **Transcripts** (CId:STRING, SId:INT, Grade:GRADES, Comment:STRING)

relationship

- Defines the relationship between relations
- A key FK in a relation R is a foreign key iff:
 - The attributes in FK matches a primary key PK of a relation S and they are of the same type
 - Any record i in R has a value in FK that either
 - occurs as a value of PK for some tuple j in S, or
 - is null
 - I.e., FK = PK (domain and values)
- A relation can have several foreign keys

Keys in Relations

Foreign Keys -- Example



All Keys in a Relation

Superkeys

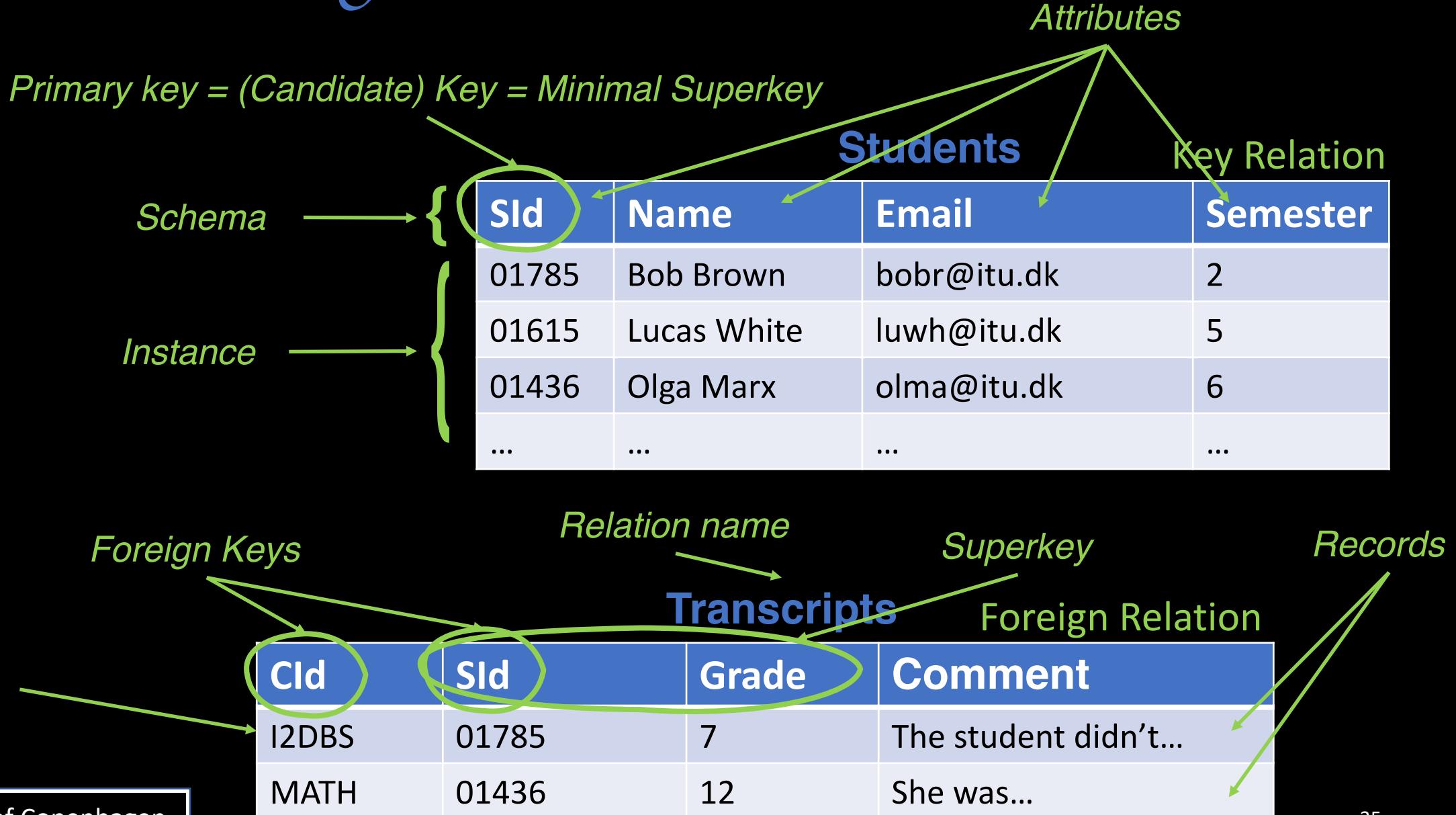
Minimal Superkeys & (Candidate) Keys



Note

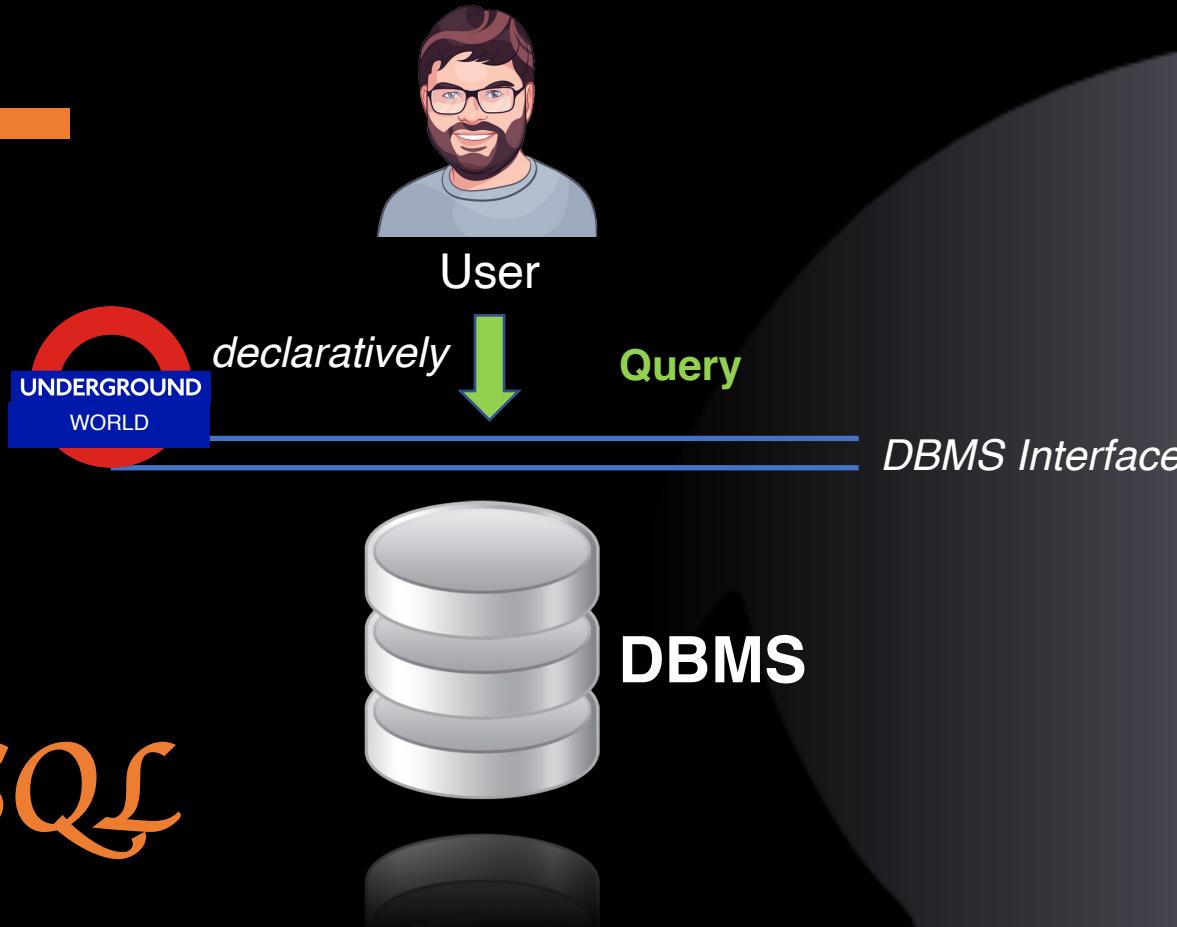
Keys are part of the schema

So Far, All Together



Integrity Constraints

- An integrity constraint (IC) is a limitation of the allowed content (or development) of a database
 - Ensures that the data are always correct and consistent
 - There exist various ICs
- It is the RDBMS that takes care of ensuring the ICs in a database
- ICs already seen so far
 - Domain constraint -- attribute type and format (e.g., DATE)
 - Key constraint -- uniqueness & minimality
 - Entity constraint (PK) -- NOT NULL
 - Referential constraint (FK) -- PK = FK
- More advanced ICs
 - Functional dependencies
 - Temporal constraint...



Readings:

PDBM 7.1-7.2

Structured Query Language (SQL)

- SEQUEL if you worked for IBM in the 80s
- SQL is primarily a query language, for getting information from a database (DML)
 - also includes a data-definition component for describing database schemas (DDL)
- Invented in the 70s by IBM
- The three most common commands in SQL queries
 - SELECT, FROM, WHERE

```
sql
SELECT * FROM Students WHERE Name = 'Lucas White';
```

History [edit]

SQL was initially developed at IBM by Donald D. Chamberlin and Raymond F. Boyce in the early 1970s.^[14] This version, initially called *SEQUEL (Structured English Query Language)*, was designed to manipulate and retrieve data stored in IBM's original quasi-relational database management system, *System R*, which a group at IBM San Jose Research Laboratory had developed during the 1970s.^[14] The acronym SEQUEL was later changed to SQL because "SEQUEL" was a trademark of the UK-based Hawker Siddeley aircraft company.^[15]

- **Data Definition Language (DDL)**
 - Used by the database administrator (DBA) to define the database's data model
 - Three common commands:
 - CREATE TABLE, ALTER TABLE, and DROP TABLE

Today's focus

- **Data Manipulation Language (DML)**
 - Used by applications and users to retrieve, insert, modify, and delete records
 - Four statements:
 - SELECT, INSERT, UPDATE, and DELETE

- **Each attribute in a relation has:**
 - a primitive type (atomic values), and;
 - an unique name
- **The main goal is to eliminate redundant data in a relation**
- **Benefits:**
 - Data integrity
 - Data consistency
 - Easy data manipulation
 - Better data organization

Type	Description
CHAR(n)	Fixed-length string of size n
VARCHAR(n)	Variable-length string of maximum size n
SMALLINT	Small integer (-32,768 and 32,767)
INT	Integer (-2,147,483,648 and 2,147,483,647)
FLOAT(n, d)	Small number with a floating decimal point: n = max digits and d = max decimals
DOUBLE(n, d)	Large number with a floating decimal point: n = max digits and d = max decimals
DATE	Date in format YYYY-MM-DD
DATETIME	Date and time in format YYYY-MM-DD HH:MI:SS
TIME	Time in format HH:MI:SS
BOOLEAN	True or false
BLOB	Binary large object (typically unstructured)

Note

Check types in PostgreSQL

SId	Name	Email	Semester

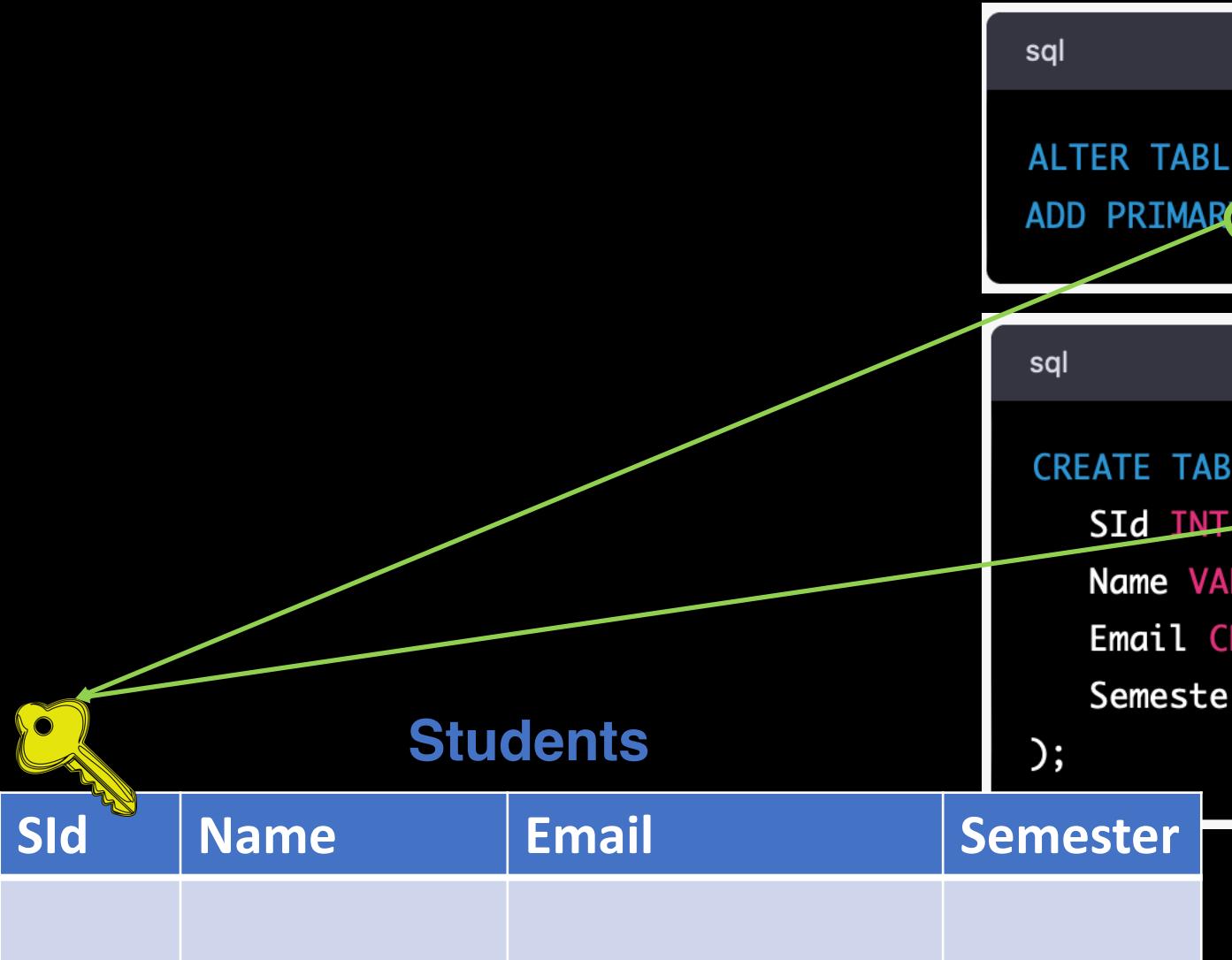
sql

```
CREATE TABLE Students (
    SId INT,
    Name VARCHAR(255),
    Email CHAR(11),
    Semester INT
);
```

Advise

Start playing with
PostgreSQL ASAP!

Define a Primary Key



Students

SId	Name	Email	Semester
			

sql

```
CREATE TABLE Departments (
    DId INT PRIMARY KEY,
    Name VARCHAR(255)
);
```

Departments

DId	Name
	

sql

```
CREATE TABLE Courses (
    CId VARCHAR(25) PRIMARY KEY,
    Name VARCHAR(255),
    DId INT,
    FOREIGN KEY (DId) REFERENCES Departments (DId)
);
```

Courses

CId	Name	DId
		

Students

SId	Name	Email	Semester
			

Courses

CId	Name	DId
		

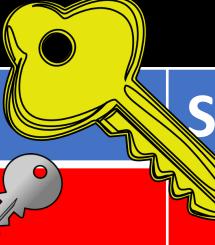
Departments

DId	Name
	

CId	SId	Grade	Comment
			

```
CREATE TABLE Transcripts (
    CId VARCHAR(25),
    SId INT,
    Grade VARCHAR(10),
    Comment VARCHAR(255),
    PRIMARY KEY (CId, SId),
    FOREIGN KEY (CId) REFERENCES Courses (CId),
    FOREIGN KEY (SId) REFERENCES Students (SId)
);
```

- **What if a value is missing?**
 - Does not exist?
 - Unknown?
 - Secret?
- **SQL solution: NULL**
 - $\text{NULL} = \text{no value}$
 - More next week...
- **By default, attributes can be NULL**
 - Except: PRIMARY KEY attributes
 - Except: NOT NULL attributes
- **Allowing NULL values is a design decision!**



Transcripts				
CId	SId	Grade	Comment	
key	key			

sql

```
CREATE TABLE Transcripts (
    CId VARCHAR(255) NOT NULL,
    SId INT NOT NULL,
    Grade VARCHAR(10) NOT NULL,
    Comment VARCHAR(255),
    PRIMARY KEY (CId, SId),
    FOREIGN KEY (CId) REFERENCES Courses(CId),
    FOREIGN KEY (SId) REFERENCES Students(SId)
);
```

- Cannot be null
- Can be null

```
sql
DROP TABLE Transcripts;
DROP TABLE Courses;
DROP TABLE Students;
DROP TABLE Departments;
```

Takeaways

- **Relational model**
 - Relations, attributes, keys, primary & foreign keys, ...
- **SQL DDL = Data Definition Language**
 - CREATE TABLE, DROP TABLE, ALTER TABLE, ...
 - Allows to create complex schemas and maintain them
- **SQL DML = Data Manipulation Language**
 - INSERT, DELETE, UPDATE, SELECT
 - Simple set of commands for complicated actions
 - (*we will dive into it next week*)



What is next?

- **Next week: SQL and SQL**
 - SQL DML Basics, joins, aggregates, grouping, ...
- **Install PostgreSQL**
 - ... if you have not already done this
 - Problems: get help during exercises!
- **Exercises today**
 1. Run DB install script and queries from lecture
 2. Create a sample “Coffee” database
 - Use commands in the remainder of these slides...
 - Write some INSERT and SELECT statements
 - Play with constraints
 3. Consider databases without a DBMS

Introduction to Database Systems

I2DBS – Spring 2023

- Week 2:
- SQL Select
- SQL Joins
- SQL Aggregations

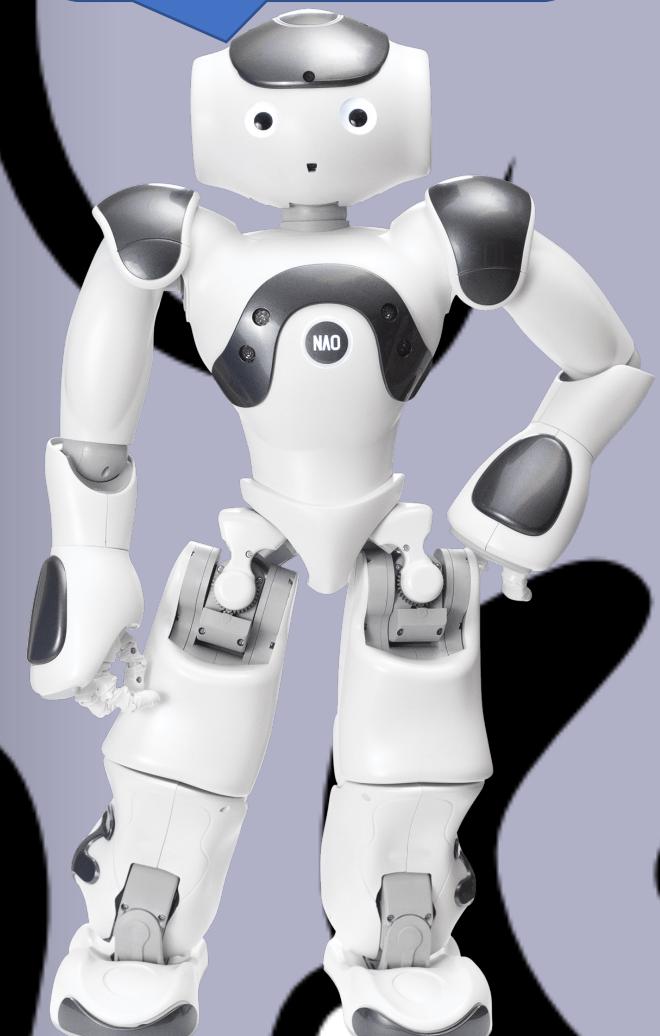
Jorge-Arnulfo Quiané-Ruiz

Readings:

PDBM 7.3

chatGPT

Hi, I will be
your virtual TA!



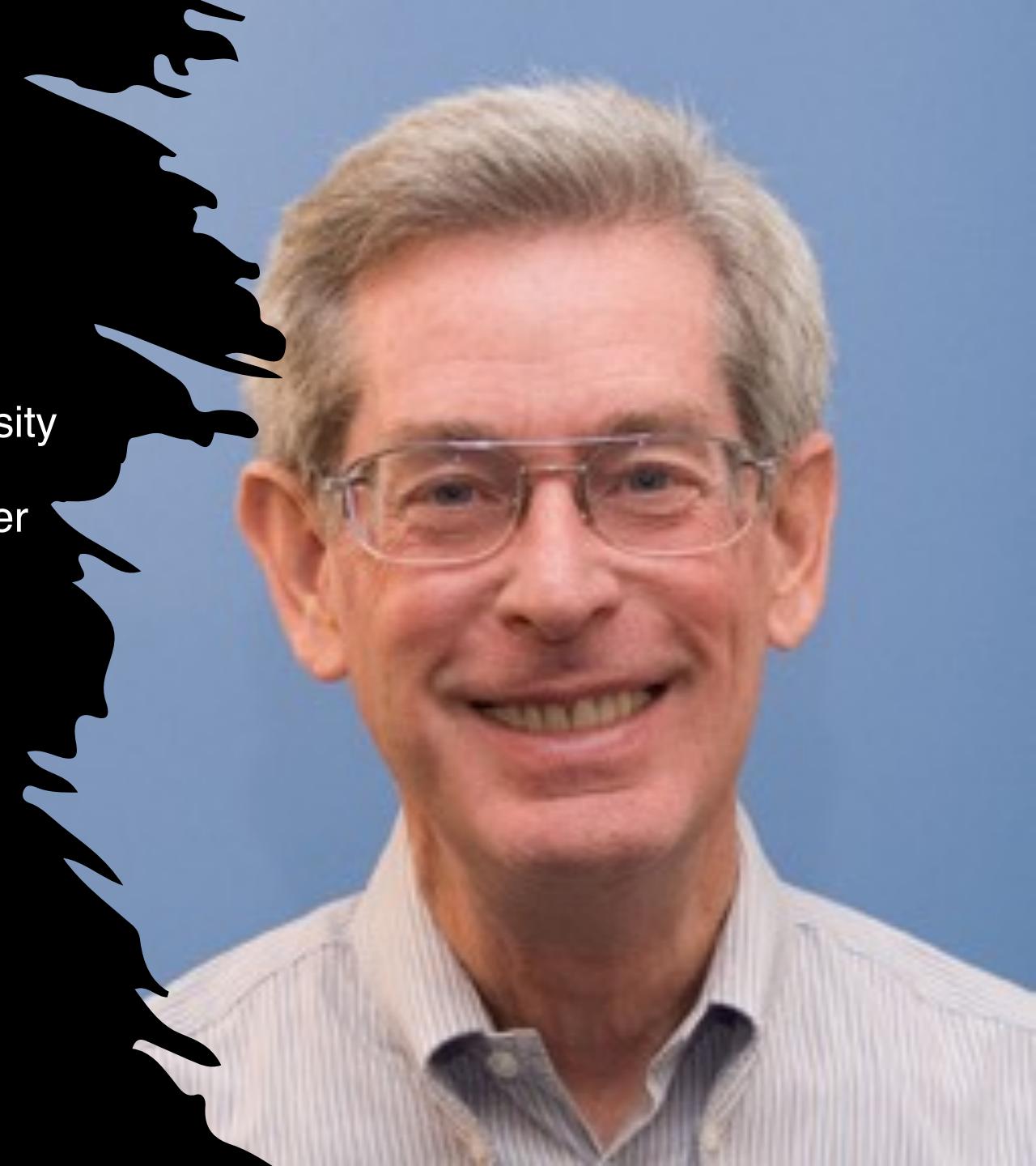


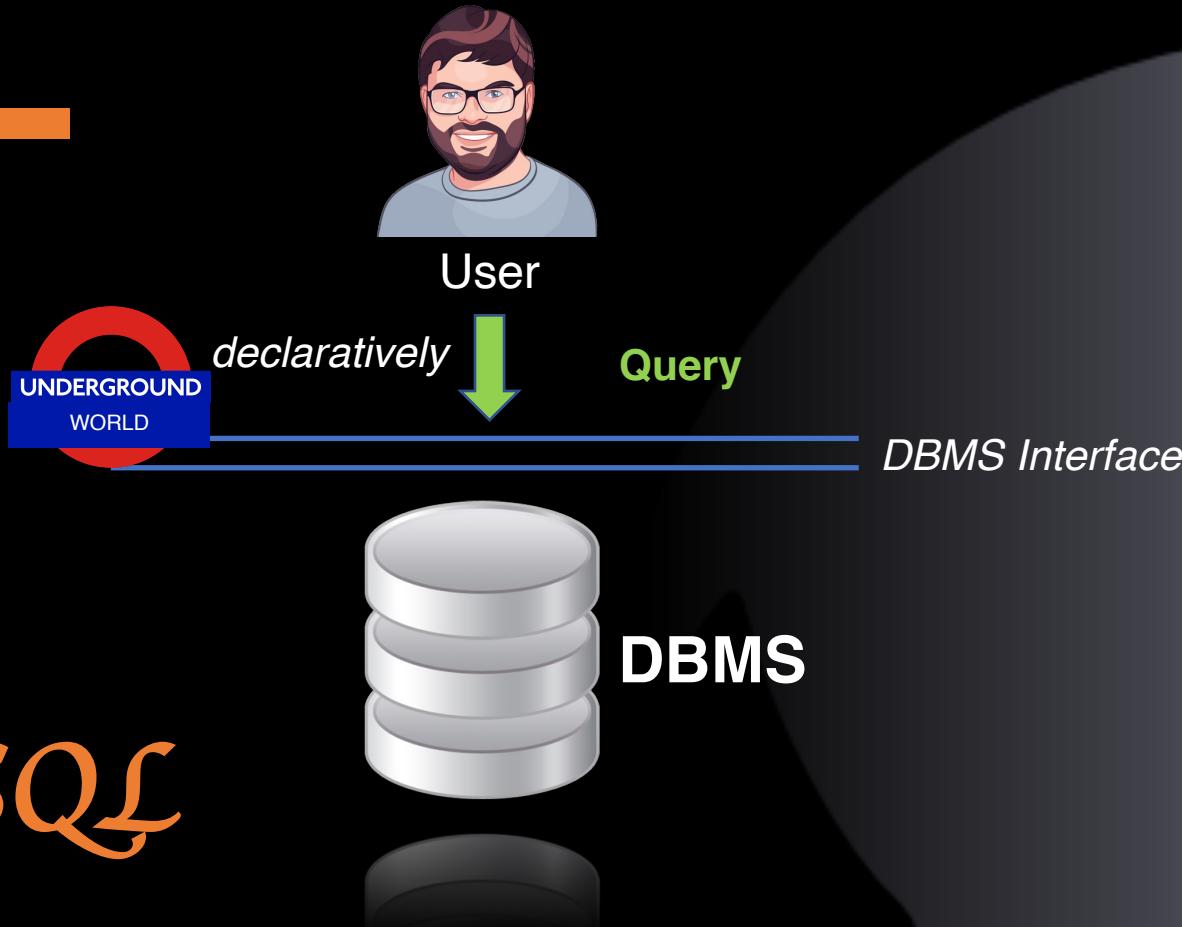
Profile of the Week

Donald D. Chamberlin

SQL Co-Inventor

- **1944:** Born in San Jose California
- **1971:** PhD in Elec. Eng. from Stanford University
- **1973:** Moved to IBM Almaden Research Center
- **Mid-1970:** Co-Invented SQL
- **1994:** ACM Fellow
- **1997:** ACM SIGMOD Innovations Award
- **2003:** IBM Fellow





Readings:

PDBM 1

- **Data Definition Language (DDL)**
 - Used by the database administrator (DBA) to define the database's data model
 - Three common commands:
 - CREATE TABLE, ALTER TABLE, and DROP TABLE

- **Data Manipulation Language (DML)**
 - Used by applications and users to retrieve, insert, modify, and delete records
 - Four statements:
 - SELECT, INSERT, UPDATE, and DELETE

Today's focus

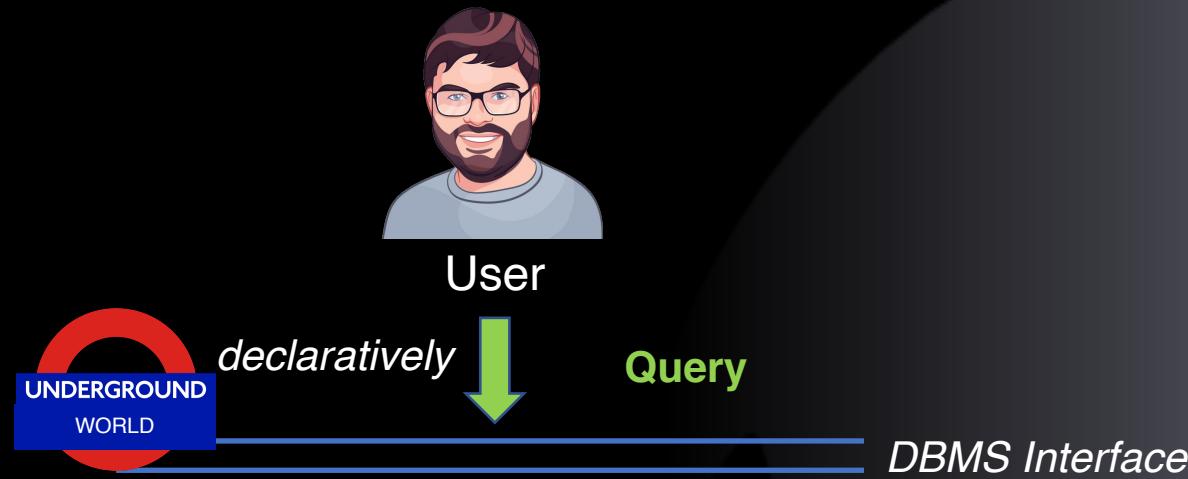
Database Schema

Running Example

- Coffees(name, manufacturer)
- Coffeehouses(name, address, license)
- Drinkers(name, address, phone)
- Likes(drinker, coffee)
- Sells(coffeehouse, coffee, price)
- Frequents(drinker, coffeehouse)

Note

Keys are underlined



SQL DML -- Select

Readings:

PDBM 1

Rationale & Example

- **SELECT** -- desired attributes
FROM -- one or more relations
WHERE -- condition about records of the relations
- Which coffees are made by Ottolina?

Coffees	
name	manufacturer

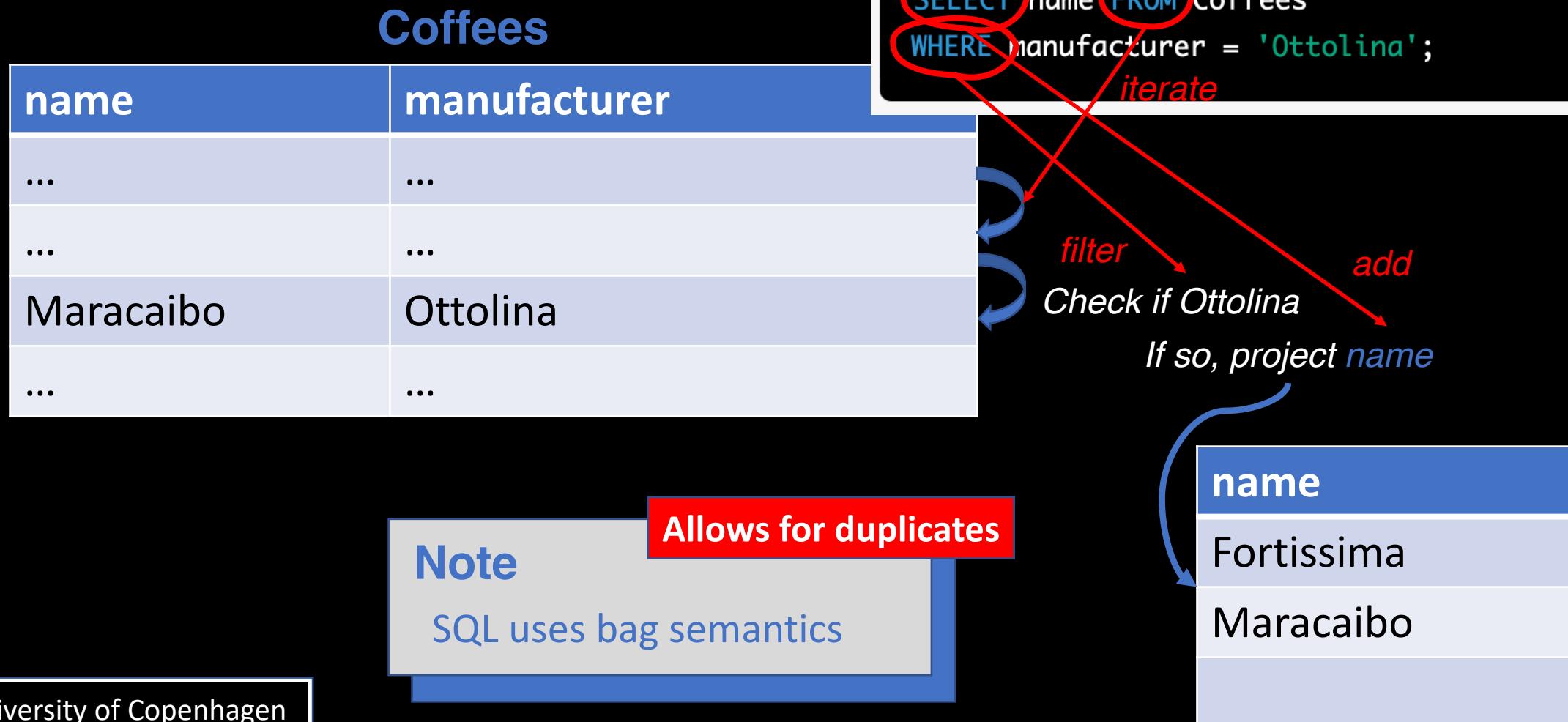
sql

```
SELECT name FROM Coffees  
WHERE manufacturer = 'Ottolina';
```



name
Fortissima
Maracaibo
Buongiorno

The Same but Graphically



Star in SELECT

- What if the **SELECT** clause has * instead?

Coffees	
name	manufacturer

```
sql
SELECT * FROM Coffees
WHERE manufacturer = 'Ottolina';
```

- This selects all attributes from the Coffees relation

name	manufacturer
Fortissima	Ottolina
Maracaibo	Ottolina
Buongiorno	Ottolina

Attribute Renaming

- What If you want the result to have different attribute name?
 - Use then the keyword AS

Coffees	
name	manufacturer

```
vbnet
SELECT name AS coffee, manufacturer FROM Coffees
WHERE manufacturer = 'Ottolina';
```

coffee	manufacturer
Fortissima	Ottolina
Maracaibo	Ottolina
Buongiorno	Ottolina

Expressions in SELECT

- Any expression that makes sense can appear as an element of a SELECT clause.

```
SELECT coffeehouse, coffee, price * 0.965 AS priceInYen FROM Sells;
```

Sells

coffeehouse	coffee	price

coffeehouse	coffee	priceInYen
Sue's	Fortissima	342
Joe's	Maracaibo	285
Bob's	Buongiorno	149

Constants as Expressions in SELECT

- Any constant as expression (example with the Likes relation)

Likes		sql	
drinker	coffee	SELECT drinker, 'likes Fortissima' AS whoLikesFort FROM Likes WHERE coffee = 'Fortissima';	
drinker		whoLikesFort	
Sally		Likes Fortissima	
Fred		Likes Fortissima	

Live Exercise

- What should the following queries return?
 - Assume the relation Coffees has two records only

coffee	manufacturer
Fortissima	Ottolina
Maracaibo	Ottolina

coffee	manufacturer
Fortissima	Ottolina
Maracaibo	Ottolina

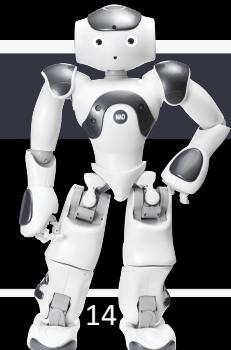
```
sql  
  
SELECT *  
FROM Coffees;
```

Query
42
42

```
vbnet  
  
SELECT 42 AS Query  
FROM Coffees
```

Query
42

```
vbnet  
  
SELECT 42 AS Query;
```



Practice

- Analog has a 50% discount on a second coffee of the same type. For each coffee sold there, show the price of two coffees ('priceOfTwo').
- A coffeehouse owner by the name of “Eleni” has passed out with caffeine shock. Write a query to find her home phone.

Conditions in WHERE

- Comparisons $=, \neq, <, >, \leq, \geq$.
 - Many other operators that produce boolean-valued results.
- Boolean operators AND, OR, NOT.
- Find the price Joe's coffeehouse charges for Maracaibo in Sells

Sells

coffeehouse	coffee	price

sql

```
SELECT price
FROM Sells
WHERE coffeehouse = 'Joe''s' AND coffee = 'Maracaibo';
```

Patterns

- A condition can compare a string to a pattern by:
 - <Attribute> LIKE <pattern>
 - <Attribute> NOT LIKE <pattern>
- Pattern is a quoted string with
 - % = “any string”
 - _ = “any character”
- Escaping characters
 - \" default escape character
 - ESCAPE modifier
- PostgreSQL supports regular expressions
 - These are much more expressive!

sql

```
SELECT * FROM Coffeehouses WHERE address LIKE '%1';
```

sql

```
SELECT * FROM Coffeehouses WHERE address LIKE '%!_1' ESCAPE '!';
```

Practice

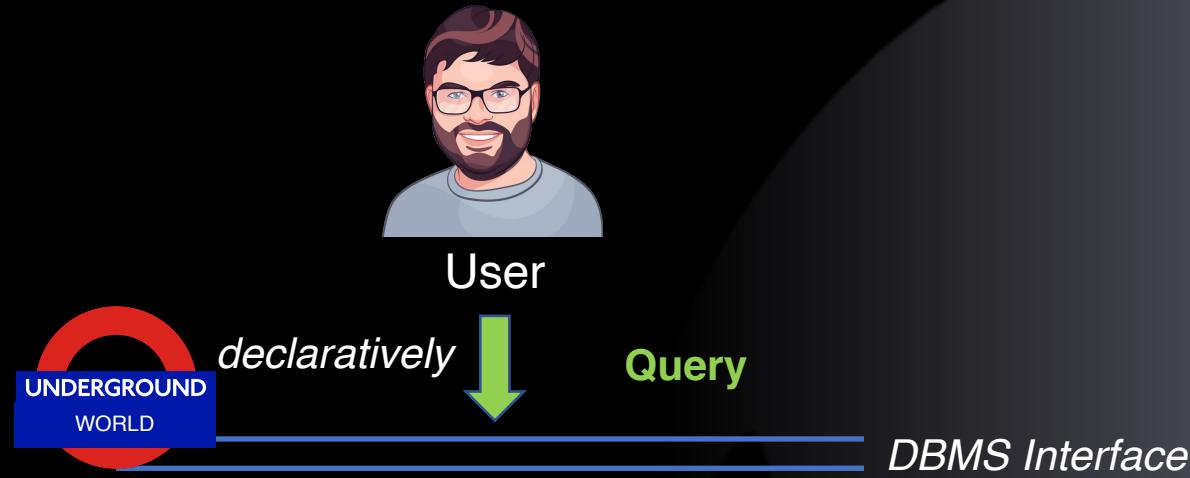
- A coffeehouse patron by the last name of “Sivertsen” has passed out, again from caffeine shock. Show a query to find his home phone.
- Someone has recommended a coffeehouse that sells a coffee called “Blue”-something, that costs more than 100. Unfortunately, she doesn’t remember the name of the coffeehouse. Find where that coffee is sold at such a high price.

Sorting

- SQL assumes bags semantics
 - No order assumption
- ORDER BY sorts the results
 - Ascending (ASC) or descending (DESC)

```
sql
SELECT coffeehouse, price
FROM Sells
ORDER BY coffeehouse, price DESC;
```

- When should we use ORDER BY?



SQL DML -- Joins

Readings:

PDBM 1

Multi-Relation Queries

- Most typically, queries combine data from more than one relation.
- Several relations in the FROM clause.
- Distinguish attributes of the same name by “<relation>.<attribute>”
- Find the coffees liked by at least one person who frequents Joe's.

sql

```
SELECT coffee
FROM Likes
JOIN Frequent
ON Likes.drinker = Frequent.drinker
WHERE Frequent.coffeehouse = 'Joe''s';
```

The Same but Graphically

sql

```
SELECT coffee
FROM Likes
JOIN Frequent
ON Likes.drinker = Frequent.drinker
WHERE Frequent.coffeehouse = 'Joe''s';
```

Likes

drinker	coffee
...	...
...	...
Sally	Maracaibo
...	...

join

Frequent

drinker	coffeehouse
...	...
...	...
Sally	Joe's
...	...

filter

Join Semantics

- Start with the product of all relations in the **FROM clause**
 - Imagine one tuple-variable for each relation in the FROM clause.
 - Think of nested for-loops
- Apply the selection conditions from the **JOIN clauses**
 - Usually F.foreign_key = K.key
- Apply the selection condition from the **WHERE clause**
- Apply the selection Project onto the list of attributes and expressions in the **SELECT clause**.

sql

```
SELECT coffee
  FROM Likes
  JOIN Frequent
    ON Likes.drinker = Frequent.drinker
   WHERE Frequent.coffeehouse = 'Joe''s';
```

Likes

Frequent

drinker	coffee
...	...
Sally	Maracaibo

drinker	coffeehouse
...	...
Sally	Joe's

Renaming Relations

sql

```
SELECT coffee
FROM Likes
JOIN Frequent
ON Likes.drinker = Frequent.drinker
WHERE Frequent.coffeehouse = 'Joe''s';
```

sql

```
SELECT L.coffee
FROM Likes L
JOIN Frequent F
ON L.drinker = F.drinker
WHERE F.coffeehouse = 'Joe''s';
```



Practice

- For each person that “frequents” some coffeehouse, show the name of the person and the address of the coffeehouse.
- For each person that “frequents” some coffeehouse, show the address of the person and the address of the coffeehouse.
- For each person that “frequents” some coffeehouse, show the name of the person and the address of the coffeehouse.
 - Why don’t we need the Drinkers relation?
 - What if we want only the names of drinkers and coffeehouses?
 - What if we used IDs in our design, and wanted the names?
 - What if we want all drinkers, including those that don’t frequent any coffeehouse?
 - What if we only want the name of drinkers?

SQL DML Rationale (recall) + Join

- **SELECT** -- desired attributes
- FROM** -- one or more relations
- WHERE** -- condition about records of the relations
- JOIN** -- connect records between relations

Duplicate Elimination

- Force the result to be a set with **SELECT DISTINCT ...**
 - Can also do this with GROUP BY – **but please don't!**
- From **Sells(coffeehouse, coffee, price)**
 - find all the different prices charged for coffees:

Note

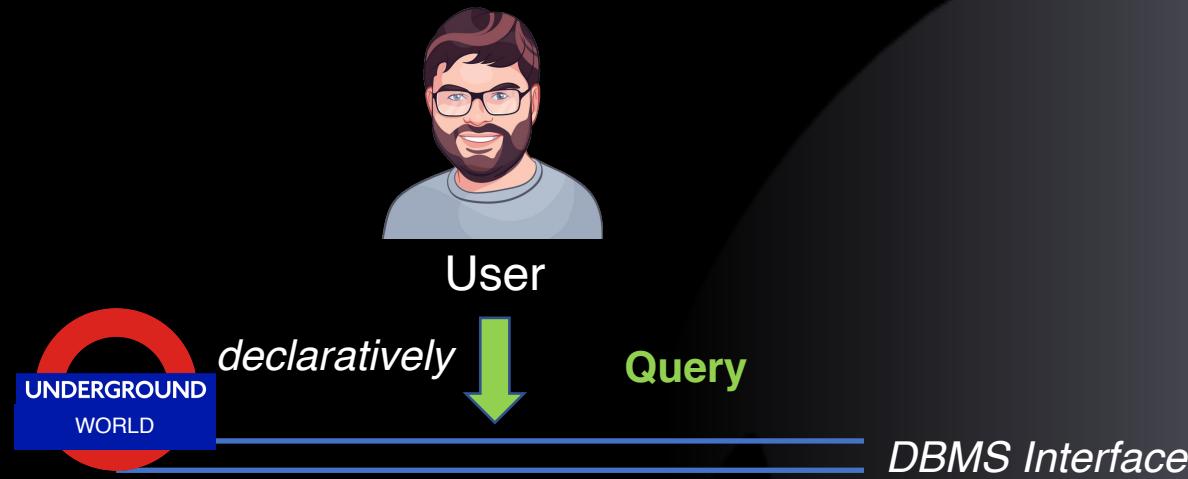
Sorting = $O(n \log n)$

sql

```
SELECT DISTINCT price  
FROM Sells;
```

Note

Without DISTINCT, each price would be listed as many times as there were coffeehouse/coffee pairs at that price



SQL DML -- Aggregations

Readings:

PDBM 1

Relation-based Aggregations

- **SUM, AVG, COUNT, MIN, and MAX can be applied to a column in a SELECT clause**
 - Produces that aggregation on the column
- **COUNT(*) counts the number of tuples.**
- **Find the average price of Maracaibo**

sql

```
SELECT AVG(price)
FROM Sells
WHERE coffee = 'Maracaibo';
```

Duplicate Elimination

- Use DISTINCT inside an aggregation.
- Find the number of different prices charged for Maracaibo

sql

```
SELECT COUNT(DISTINCT price)
FROM Sells
WHERE coffee = 'Maracaibo';
```

Practice

- Show the highest price of any coffee

Nulls in Aggregations

Sells

coffeehouse	coffee	price

- **NULL never contributes to a sum, average, or count, and can never be the minimum or maximum of a column.**
- **But if there are only NULL values in a column**
 - The result of the aggregation is NULL
- **Exception: COUNT of an empty set is 0.**

Note

The number of coffeehouses
at a known price!

sql

```
SELECT COUNT(coffeehouse)
FROM Sells
WHERE coffee = 'Maracaibo';
```

sql

```
SELECT COUNT(price)
FROM Sells
WHERE coffee = 'Maracaibo';
```

Grouping-based Aggregations

- **SELECT-FROM-WHERE expression followed by GROUP BY**
- **The relation that results from the SELECT-FROM-WHERE is:**
 - grouped according to the values of the attributes in the GROUP BY clause
 - any aggregation is applied only within each group
- **How does the system perform GROUP BY?**
 - sort the relation OR hash it!
- **Find the average price for each coffee**

sql

```
SELECT coffee, AVG(price)
FROM Sells
GROUP BY coffee;
```

coffee	AVG(price)
Maracaibo	2.33
Fortissima	3.51

Grouping-based Aggregations

- Find for each drinker the average price of Maracaibo at the coffeehouses they frequent

```
vbnet
SELECT drinker, AVG(price)
FROM Frequent F
JOIN Sells S ON F.coffeehouse = S.coffeehouse
WHERE coffee = 'Maracaibo'
GROUP BY drinker;
```

Frequent

drinker	coffeehouse

Sells

coffeehouse	coffee	price

SELECT Attributes with Aggregations

- If any aggregation is used, then each element of the SELECT list must be either:
 - An attribute on the GROUP BY list
 - Aggregation – COUNT, AVG, MAX, ...
- All GROUP BY attributes must be in SELECT
 - Caveat: SQL standard vs. Implementation
 - Some systems are more flexible!
 - PostgreSQL allows omitting functionally dependent attributes (see week 6!)

Example Query

coffeehouse	coffee	price

- For each coffeehouse, show the average price of all coffees in that coffeehouse.

vbnet

```
SELECT coffeehouse, AVG(price) AS average_price  
FROM Sells  
GROUP BY coffeehouse;
```



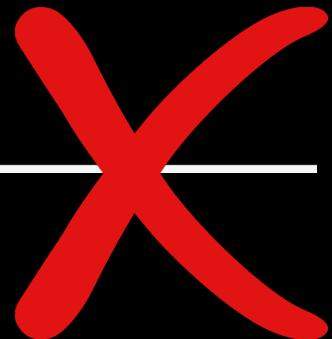
coffeehouse	average_price

What About this One?

- Show the price of the most expensive coffee (from any coffeehouse) and the name of the coffeehouse that sells it

sql

```
SELECT coffeehouse, MAX(price)  
FROM Sells  
GROUP BY coffeehouse;
```



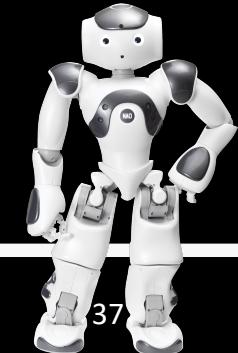
sql

```
SELECT coffeehouse, MAX(price)  
FROM Sells
```



sql

```
SELECT coffeehouse, price  
FROM Sells  
WHERE price = (SELECT MAX(price) FROM Sells);
```



Practice

- For each coffeehouse, show the number of coffees sold in that coffeehouse and the average price.
- For each coffeehouse, show the price of the most expensive coffee sold in that coffeehouse.
- Show the name and price of the least expensive coffee (from any coffeehouse).

HAVING Clause

- HAVING <condition> may follow a GROUP BY clause.
 - If so, the condition applies to each group, and groups not satisfying the condition are eliminated.
- Like WHERE but for groups!
- Find the average price of those coffees that are served in at least two coffeehouses

Sells

coffeehouse	coffee	price

```
sql
SELECT coffee, AVG(price)
FROM Sells
GROUP BY coffee
HAVING COUNT(coffeehouse) > 1
```

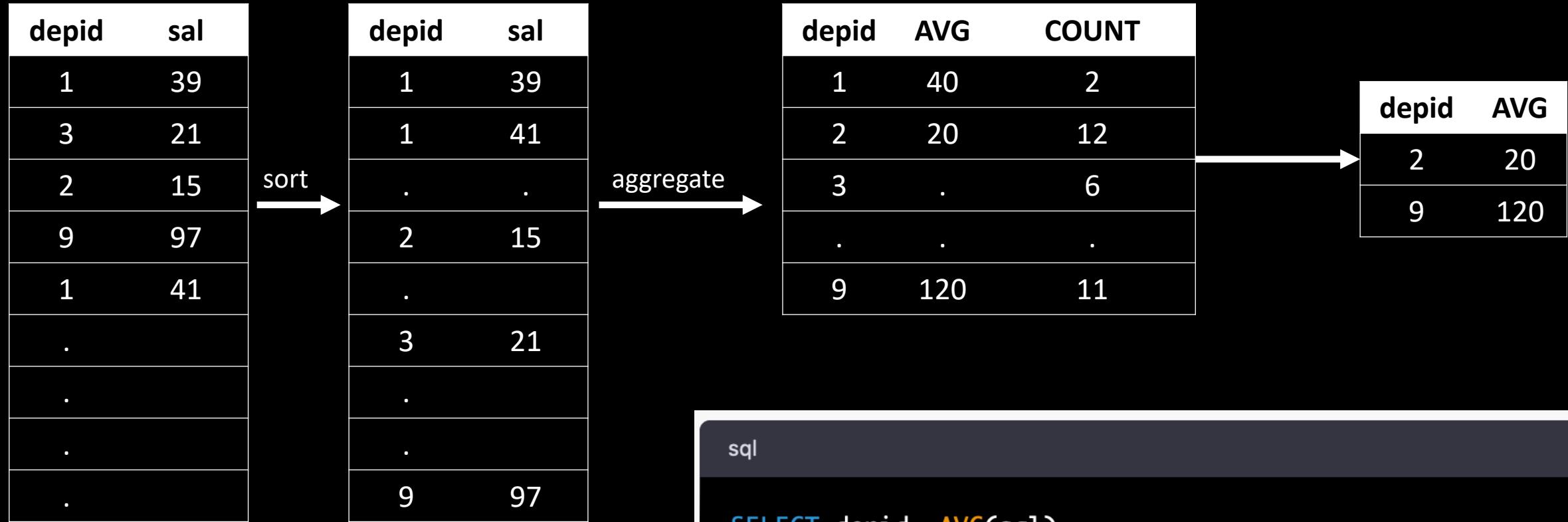
HAVING Conditions

- Anything goes in a sub-query – later...
- Outside sub-queries, they may refer to attributes only if they are either
 - A grouping attribute, or
 - Aggregated(same condition as for SELECT clauses with aggregation)

Practice

- For each coffeehouse that sells more than two coffees, show the number of coffees sold in that coffeehouse and their average price.
- For each coffeehouse, show the number of drinkers that frequent the coffeehouse.
- For each drinker that frequents more than one coffeehouse, show the number of coffeehouses that he/she frequents.

GROUP BY + HAVING Implementation



sql

```
SELECT depid, AVG(sal)
FROM Emp
GROUP BY depid
HAVING COUNT(*) > 10
```

Takeaways

- **SQL universal relational query language**
 - Result set (bag) is a table
- **1 Block**
 - select, from, where, group by, having, sort by clauses
- **Joins**
 - Combine data from many tables
- **Aggregations**
 - Group data
- **SQL is code**
 - Treat it accordingly

What is next?

- 
- **Next week: Advanced SQL**
 - **Exercise 2 is out (SQL)**
 - Remember: No submission – but very important to do it!
 - It has some queries that you did not learn how to do yet – just think about potential solutions for now, we will cover them next week.
 - **Homework 1 will be open in the next two days**
 - More complex SQL queries, quiz on LearnIT + SQL script
 - **For help:** Talk to TAs in exercise class
 - Due two weeks after opening at 23:59 (date will be specified in learnIT and Piazza) – individual submissions!
 - **Remember:** We only accept and give feedback to HWs submitted before the deadline.
 - **Do the exercises before the homework!**

Introduction to Database Systems

I2DBS – Spring 2023

- Week 3:
- Division
- More about Joins & Nulls
- Set Operation
- Subqueries
- Views

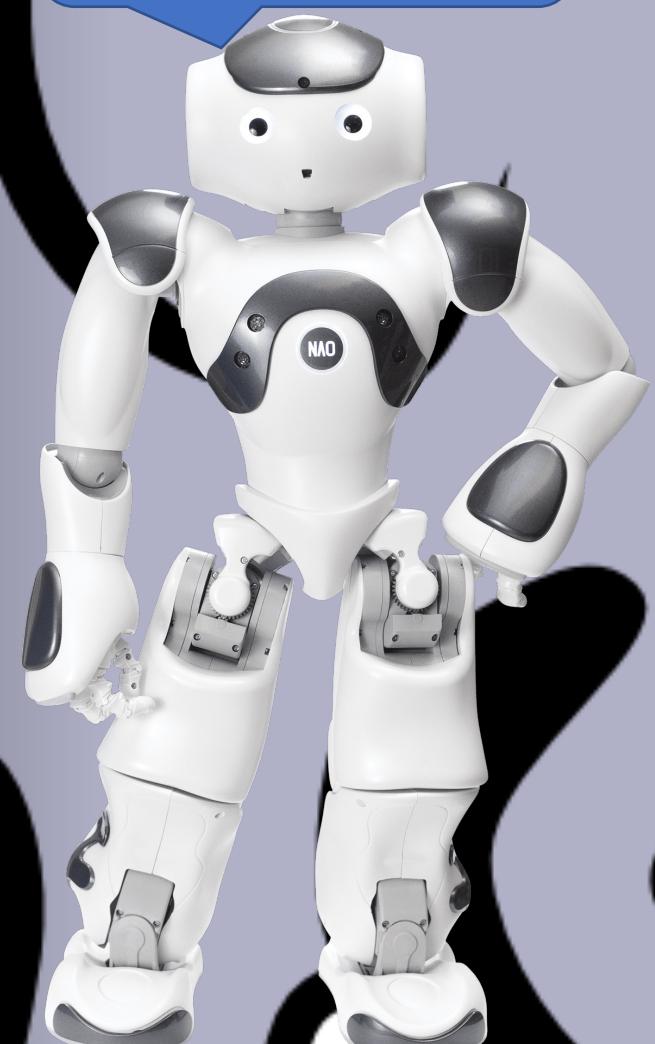
Jorge-Arnulfo Quiané-Ruiz

Readings:

PDBM 7.3, 7.4

chatGPT

Hello, welcome
back everyone!



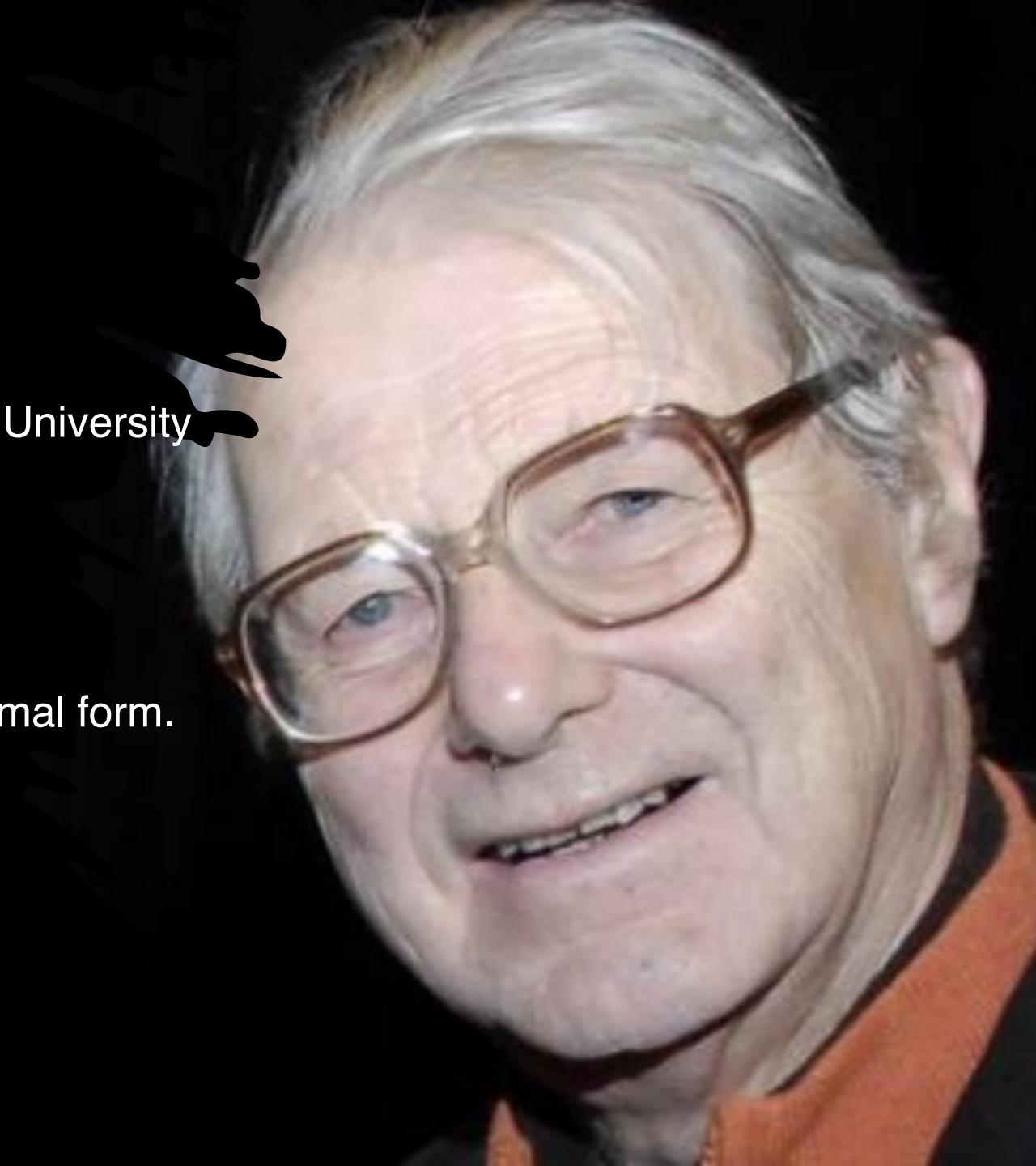


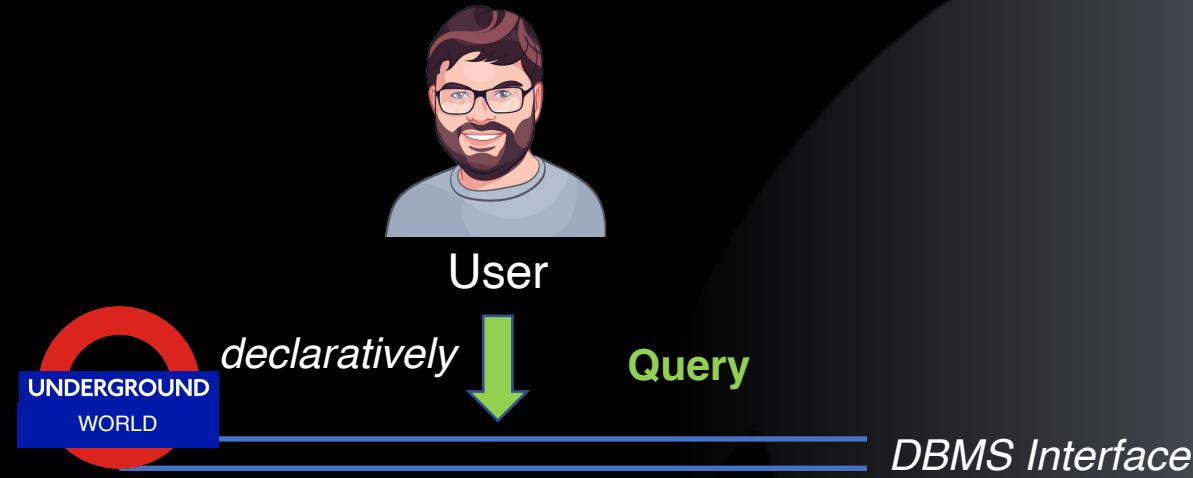
Profile of the Week

Raymond F. Boyce

SQL & BCNF Co-Inventor

- **1946:** Born in San Jose California
- **1972:** PhD in Computer Science from Purdue University
- **1972:** Moved to IBM
- **Mid-1970:** Co-Invented SQL
- **Mid-1974:** Co-developed the Boyce-Codd normal form.
- **1974:** R.I.P.





SQL - Part 2

Readings:

PDBM 1

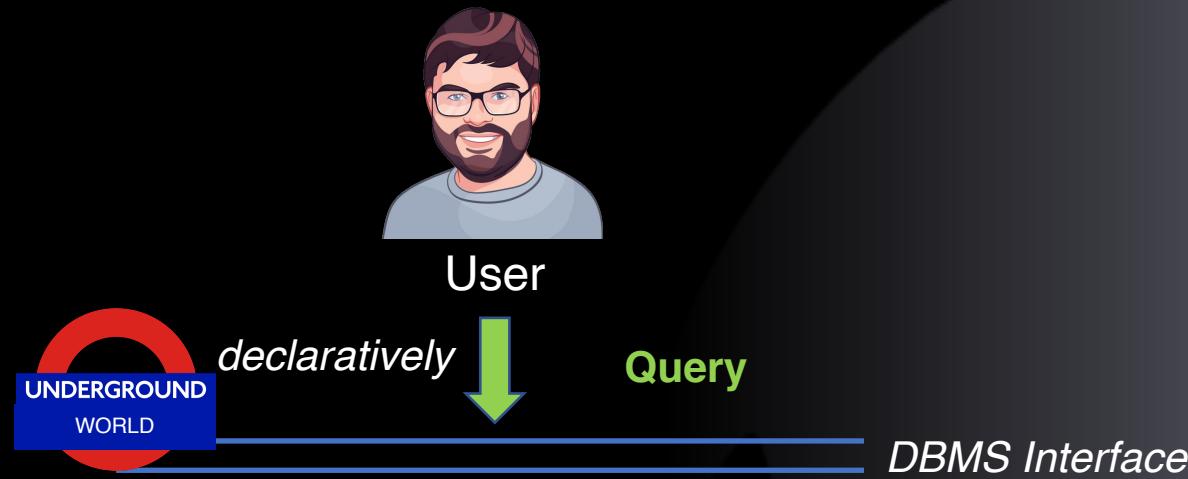
Database Schema

Running Example

- **Coffees(name, manufacturer)**
- **Coffeehouses(name, address, license)**
- **Drinkers(name, address, phone)**
- **Likes(drinker, coffee)**
- **Sells(coffeehouse, coffee, price)**
- **Frequents(drinker, coffeehouse)**

Note

Scripts on learnIT (in Week 3)



SQL DML -- Division

Readings:

PDBM 1

What is a Division?

- **R1 / R2 = records of R1 associated with all tuples of R2**
- **Find the students who have taken all courses in a program**
- **Find the airlines who land at all airports in a country/continent/the world**
- **Why divisions are not so simple in SQL?**

Division with Counting

- Find the coffeehouses that sell all existing coffees
- We can write division using GROUP BY, HAVING and a COUNT sub-query
 - Step 1: Count the number of coffees
 - Step 2: For each coffeehouse, return it only if it sells that many coffee types

```
sql
SELECT coffeehouse
FROM Sells
GROUP BY coffeehouse
HAVING COUNT(coffee) = (
    SELECT COUNT(*)
    FROM Coffees
);
```

Example Query

- Names of drinkers who frequent all coffeehouses

sql

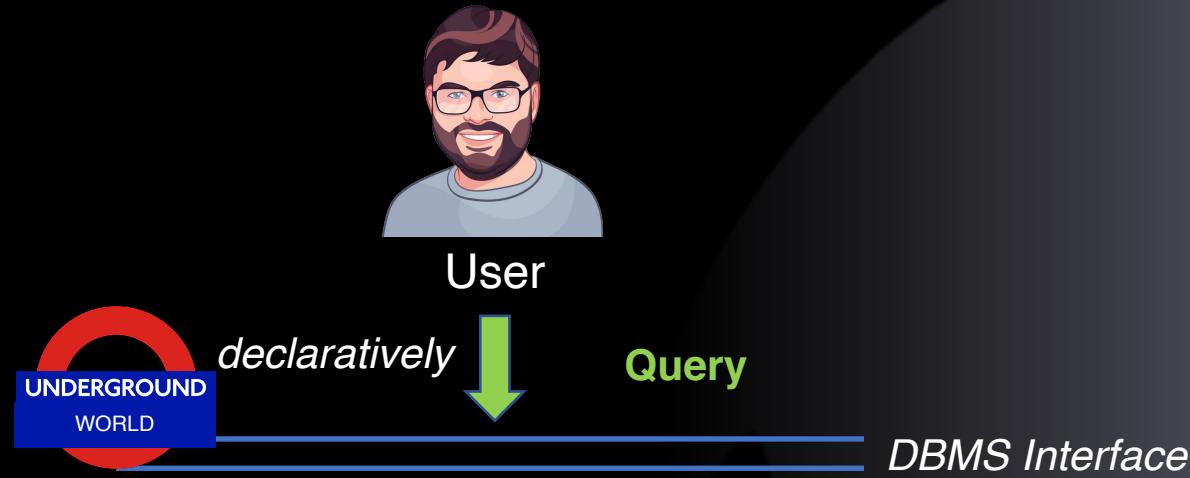
```
SELECT drinker
FROM Frequent
GROUP BY drinker
HAVING COUNT(DISTINCT coffeehouse) = (
    SELECT COUNT(*)
    FROM Coffeehouses
);
```

Example Complex Query

- Show the ID, name, record and worst result of all sports that have at least one result from every 'place' where a competition has ever been held

sql

```
SELECT S.ID, S.name, S.record, MIN(R.result)
FROM Sports S
JOIN Results R ON S.ID = R.sportID
JOIN Competitions C ON R.competitionID = C.ID
GROUP BY S.ID
HAVING COUNT(DISTINCT C.place) = (
    SELECT COUNT(DISTINCT C.place)
    FROM Competitions C
);
```



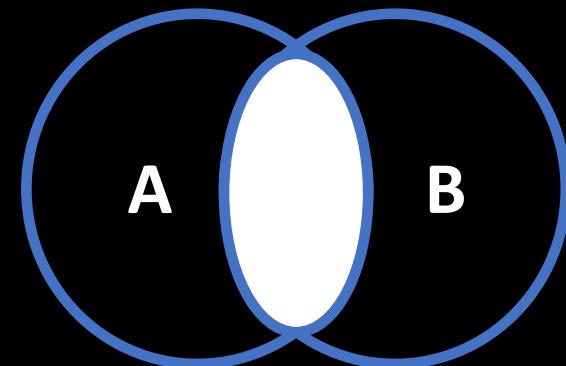
SQL DML -- Joins & Nulls

Readings:

PDBM 1

Inner Join

- R [INNER] JOIN S ON <condition>
- Example: using **Drinkers(name, address, phone)** and **Frequents(drinker, coffeehouse)**:
 Drinkers **JOIN** Frequents **ON** name = drinker;
 - gives us all (d, a, p, d, b) quintuples such that drinker d lives at address a, has phone number p, and frequents bar b

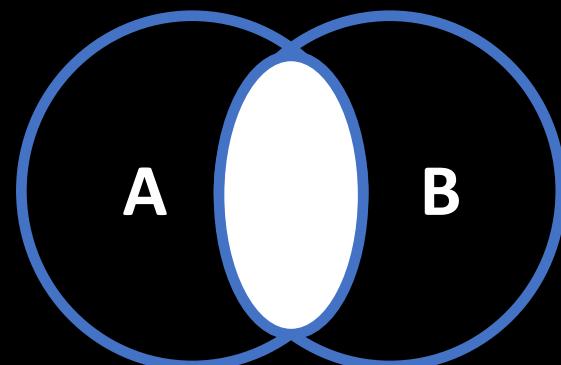


Natural Join & Cross Join (Products)

- **Natural join: R NATURAL JOIN S;**
 - Assumes “=” on all column(s) of same name
 - Removes the duplicate column(s)
 - Can be quite dangerous, so don’t use!
 - Example: Coffeehouses named by drinkers (or vice versa)
Drinkers JOIN Coffeehouses
- **Cross product: R CROSS JOIN S;**
 - Removes no columns, has no join condition
 - Can simulate with an always-true join condition

```
SELECT *
  FROM Likes L JOIN Drinkers D ON 1 = 1
```

- Rarely (but not never) the right thing to do!



Join Syntax

- We have used **ANSI join syntax**
- An alternative might be called “old-style” syntax:

```
SELECT *  
FROM Likes L  
    JOIN Drinkers D  
    ON L.drinker = D. name;
```

```
SELECT *  
FROM Likes L, Drinkers D  
WHERE L.drinker = D. name;
```

- Using **ANSI syntax is recommended!**
 - More readable, harder to forget join conditions

```
SELECT *  
FROM Likes L JOIN Drinkers D;
```

```
SELECT *  
FROM Likes L, Drinkers D;
```

Advise

If you go “old-style” then put the JOIN conditions first!

Self-Join

- From **Coffees(name, manufacturer)**, find all pairs of coffees by the same manufacturer.
- This is an example of a self-join!
 - We need to compare two coffee records with each other, so we need a "double loop" through the same relation!

Coffees		Coffees	
name	manufacturer	name	manufacturer
Maracaibo	Maracaibo Export	Maracaibo	Maracaibo Export
Fortissima	Maracaibo Export	Fortissima	Maracaibo Export



- Do not produce pairs like (Maracaibo, Maracaibo).
- Produce pairs in alphabetic order, e.g. (Fortissima, Maracaibo), not (Maracaibo, Fortissima).

```
vbnet
SELECT C1.name, C2.name
FROM Coffees C1
JOIN Coffees C2 ON C1.manufacturer = C2.manufacturer
WHERE C1.name < C2.name;
```

Example Query

- Show all coffees that are more expensive than some other coffee sold at the same coffeehouse

```
sql
SELECT S1.coffee
FROM Sells S1
WHERE S1.price > (
    SELECT MIN(S2.price)
    FROM Sells S2
    WHERE S2.coffeehouse = S1.coffeehouse
    AND S2.coffee <> S1.coffee
);
```

- Add one record and run again:

```
INSERT INTO Sells( coffeehouse, coffee, price )
VALUES( 'Mocha', 'Kopi Luwak', 400 );
```

Null-Values

- Tuples in SQL relations can have NULL as a value for one or more components.
- Meaning depends on context. Two common cases:
 - Missing value: e.g., we know Joe's Bar has some address, but we don't know what it is.
 - "secret", "figure not available", "to be announced", "impossible to calculate", "partly unknown", "uncertain", "pending"
 - Inapplicable: e.g., the value of attribute spouse for an unmarried person.
 - "undefined", "moot", "quantum uncertain", "irrelevant", "none", "n/a"

Comparing NULL Values

- The logic of conditions in SQL is really 3-valued logic: TRUE, FALSE, UNKNOWN.
- Comparing any value (including NULL itself) with NULL yields UNKNOWN.
 - $\text{NULL} \neq \text{NULL}$
 - Must use IS NULL or IS NOT NULL
- A record is in a query answer iff the WHERE clause is TRUE (not FALSE or UNKNOWN).

Example Queries

- Show the coffeehouses that sell a coffee at an unknown price

sql

```
SELECT DISTINCT coffeehouse
FROM Sells
WHERE price IS NULL;
```

- Show the name of all coffees that are sold at a known price

sql

```
SELECT DISTINCT coffee
FROM Sells
WHERE price IS NOT NULL;
```

Three-Valued Truth Tables

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

=	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	TRUE	NULL
NULL	NULL	NULL	NULL

Three-Valued Logic Example

- In two-valued logic, $p \text{ OR NOT } p$ is always TRUE
- Compute the truth value of $p \text{ OR NOT } p$ when p is UNKNOWN

Sells		
coffeehouse	coffee	price
Joe's	Maracaibo	NULL

```
sql
SELECT coffeehouse
FROM Sells
WHERE price < 2.00 OR price >= 2.00;
          ←→          ←→
          UNKNOWN      UNKNOWN
          ←→
          UNKNOWN
```



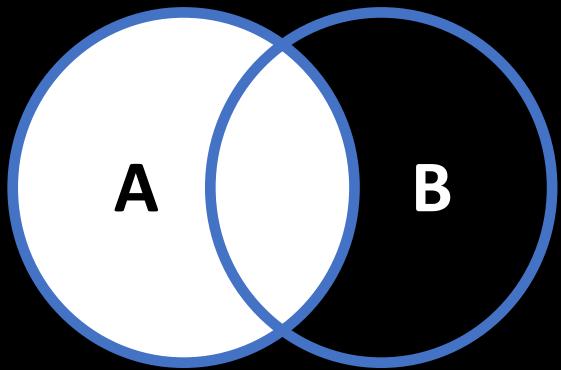
Outer Joins

- Sometimes a coffee is sold nowhere, but we want it in our result → outer join!
- R OUTER JOIN S is the core of an outer join expression.
- It is modified by optional LEFT, RIGHT, or FULL before OUTER.
 - LEFT = pad dangling tuples of R with NULL.
 - RIGHT = pad dangling tuples of S with NULL.
 - FULL = pad both; this choice is the default.

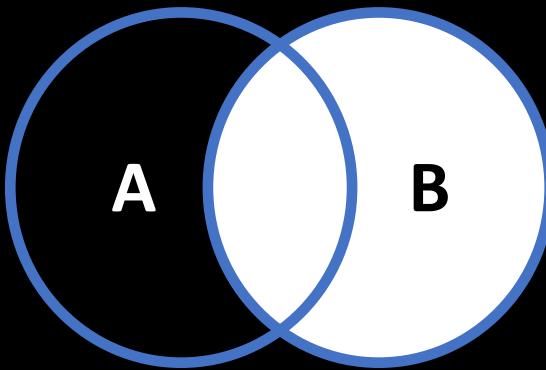
Joins in a Nutshell

vbnet

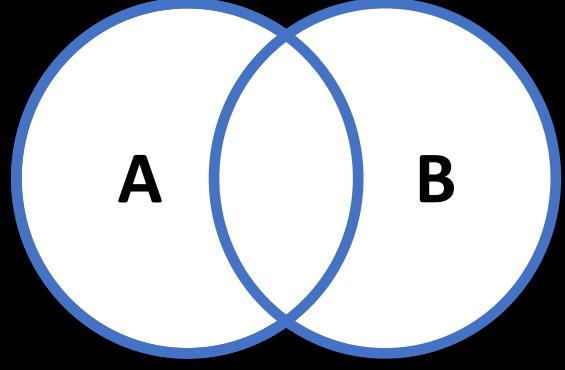
```
SELECT X.a, Y.b  
FROM X  
JOIN Y ON X.a = Y.a;
```



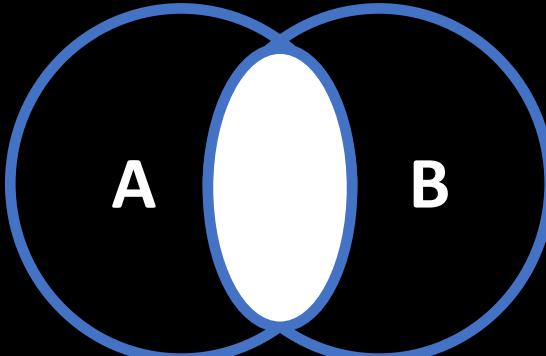
Left Outer



Right Outer



Full Outer



Inner Join

Example of an Outer Join

- Suppose we add a new coffee:

```
insert into Coffees values  
('Bragakaffi', 'Ó.J. & Kaaber')
```

```
select S.coffeehouse, C.manufacturer  
from Sells S full outer join Coffees C  
on S.coffee = C.name
```



Then the outer join includes the new coffee

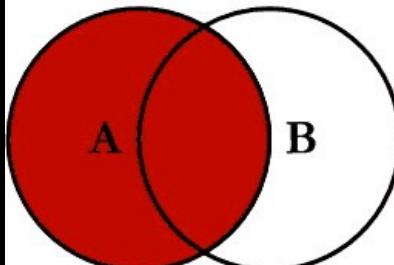
Result	
coffeehouse	manufacturer
Joe's	Ottolina
Joe's	Kopi Luwak Dir.
Sue's	Ottolina
Sue's	Marley Coffee
Sue's	Kopi Luwak Dir.
NULL	Ó.J. & Kaaber

Practice

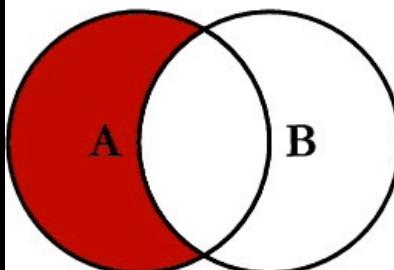
- Show all drinkers and the coffees they like, but include drinkers that do not like any coffees
- Can you use OUTER JOIN to show ONLY drinkers who do not like any coffees?

SQL Joins in a Nutshell

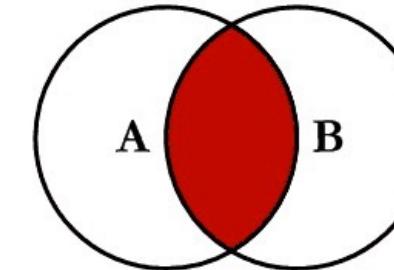
SQL JOINS



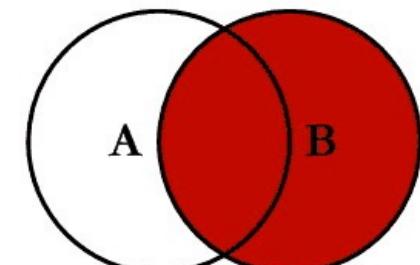
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



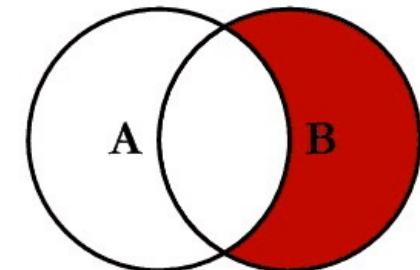
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



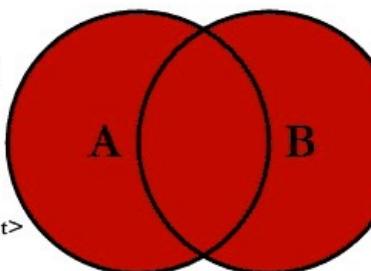
```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```

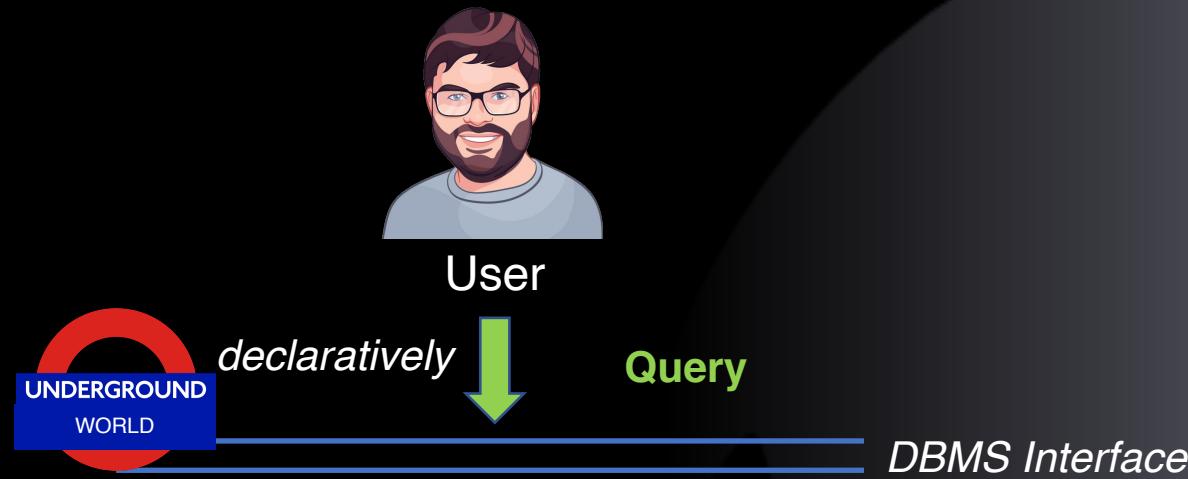


```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

© C.L. Moffatt, 2008



SQL DML – Set Operations

Readings:

PDBM 1

Set Queries

- **Syntax:**
`<Query 1>`
`UNION / INTERSECT / EXCEPT`
`<Query 2>`
- **Queries must be “union compatible” = results have matching schema:**
 - Same number of attributes
 - Attributes i of both tables have same (matching) type

Example Query -- UNION

- Show all drinkers that like “Kopi luwak” or live in “Amager”

sql

```
SELECT L.drinker
FROM Likes L
WHERE L.coffee = 'Kopi Luwak'
UNION
SELECT D.name
FROM Drinkers D
WHERE D.address = 'Amager';
```

Example Query -- *INTERSECT*

- Show all coffees that are manufactured by “Marley Coffee” and sold at an unknown price

sql

```
SELECT C.name
FROM Coffees C
WHERE C.manufacturer = 'Marley Coffee'
INTERSECT
SELECT S.coffee
FROM Sells S
WHERE S.price IS NULL;
```

Example Query - EXCEPT (occasionally MINUS)

- Show all coffeehouses that are at ‘Amager’ but which do not sell a coffee with an unknown price

sql

```
SELECT H.name
FROM Coffeehouses H
WHERE H.address = 'Amager'
EXCEPT
SELECT S.coffeehouse
FROM Sells S
WHERE S.price IS NULL;
```

Duplicates in Set Queries

- Set operators remove duplicates

R	S
1	0
2	2
2	2
3	2
3	3
4	5

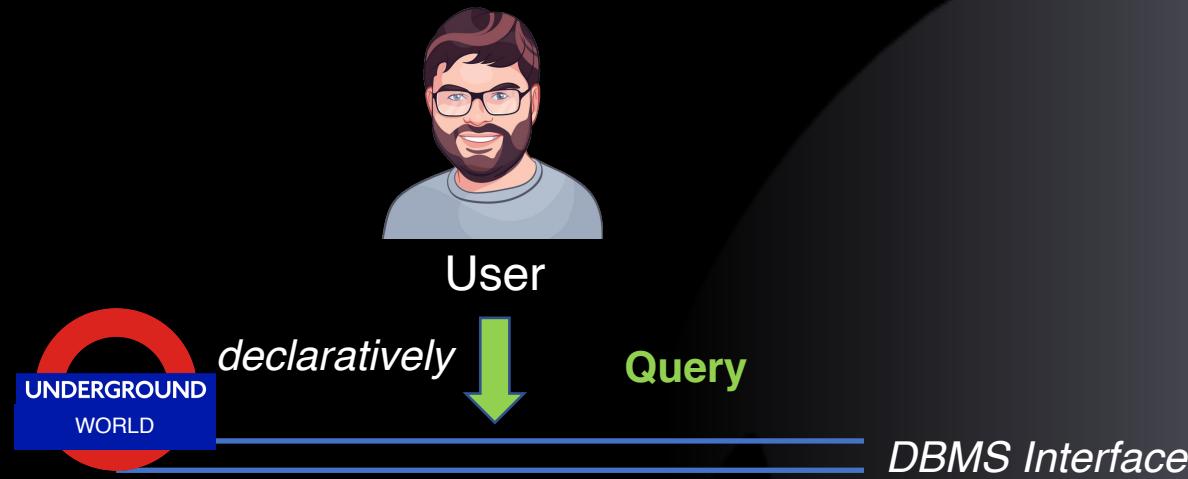
- To keep duplicates use:

- R UNION ALL S – (0, 1, 5x2, 3x3, 4, 5)
- R INTERSECT ALL S – (2x2, 1x3)
- R EXCEPT ALL S – (1, 0x2, 1x3, 4)

- Assume R has m rows and S has n rows with some values, how many rows could each at most contain?

- The most practical use:

UNION ALL when R and S are known to be disjoint!



SQL DML -- Subqueries

Readings:

PDBM 1

What is a Subquery?

- A parenthesized SELECT-FROM-WHERE statement (*subquery*) can be used as a value in a number of places, including WHERE and FROM clauses.
- If a sub-query is guaranteed to produce one tuple, then the sub-query can be used as a value
 - Usually, the tuple has only one attribute
 - An implicit type cast is performed, and a run-time error occurs if there is no tuple or more than one tuple
- A bit UGLY but practical!
 - Finding records with highest values
 - Division queries

Example of a Single-Tuple Subquery

sql

```
SELECT Coffee  
FROM Sells  
WHERE price = (SELECT min(price) FROM Sells);
```

Practice

- Find the coffeehouses that serve some coffee for the same price Mocha charges for Blue Mountain
- Makes the following two queries a single-query:
 - Find the price Mocha charges for Blue Mountain
 - Find the coffeehouses that serve a coffee at that price

Join or Subquery?

- Using a self join, find the coffeehouses that serve any coffee for the same price as Mocha charges for Blue Mountain
 - Harder to read!
 - Self-join is better for producing all pairs...

sql

```
SELECT DISTINCT S1.coffeehouse
FROM Sells S1, Sells S2
WHERE S1.price = S2.price
    AND S2.coffeehouse = 'Mocha'
    AND S2.coffee = 'Blue Mountain';
```

The IN Operator

- <tuple> **IN** (<subquery>) is true if and only if the tuple is a member of the relation produced by the subquery.
 - Opposite: <tuple> NOT IN (<subquery>)
- IN-expressions can appear in WHERE clauses.

Examples: IN

- Using Coffees(name, manufacturer) and Likes(drinker, coffee), find the name and manufacturer of each coffee that Fred likes.

sql

```
SELECT *
FROM Coffees
WHERE name IN (
    SELECT coffee
    FROM Likes
    WHERE drinker = 'Fred'
);
```

Examples: NOT IN

- Show the name of all coffeehouses that no one frequents!

sql

```
SELECT H.name  
FROM Coffeehouses H  
WHERE H.name NOT IN (  
    SELECT F.coffeehouse  
    FROM Frequents F  
)
```

Practice

- Show all drinkers that like “Kopi luwak” or live in “Amager”
- Show all coffees that are manufactured by “Marley Coffee” and sold at an unknown price
- Show all coffeehouses at ‘Amager’ which don’t sell a coffee with an unknown price

```
SELECT a  
FROM R  
WHERE b IN (  
    SELECT b  
    FROM S  
)
```

vbnet

```
SELECT a  
FROM R  
JOIN S ON R.b = S.b;
```

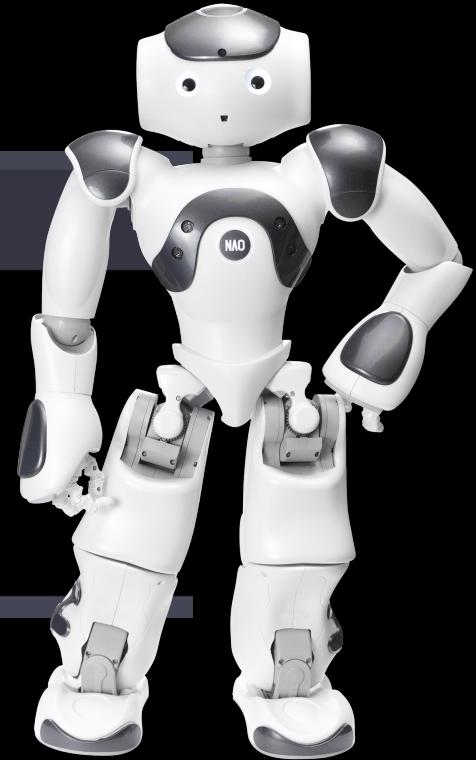


What is the Difference?

```
SELECT a  
FROM R  
WHERE b IN (  
    SELECT b  
    FROM S  
)
```

vbnet

```
SELECT DISTINCT a  
FROM R  
JOIN S ON R.b = S.b;
```



Now There is no Difference!

EXIST Operator

- **EXISTS(<subquery>)** is true if and only if the sub-query result is not empty.
- **Example:** From Coffees(name, manufacturer), find those coffees that are the only coffee by their manufacturer.

Note

Unlike IN, the sub-query is normally correlated with the outer query!

sql

```
SELECT C1.name  
FROM Coffees C1  
WHERE NOT EXISTS (  
    SELECT *  
    FROM Coffees C2  
    WHERE C2.manufacturer = C1.manufacturer  
    AND C2.name <> C1.name  
);
```

Division in Tuple Relational Calculus (TRC)

- The best method: counting!
 - Already did this – strongly recommended!!!
- In TRC – for some, the most logical method:
 - Universal quantifier – simple but does not translate to SQL
 - Double negation – “medium” complex but does translate to SQL
- Names of coffeehouses that sell at least one coffee
 - all

$$\{ R \mid \exists H \in \text{Coffeehouses} \, \forall C \in \text{Coffees} \, \exists S \in \text{Sells} \, ($$

$$H.name = S.coffeehouse \wedge$$

$$C.name = S.coffee \wedge$$

$$R.name = H.name) \}$$

Division in TRC with Double Negation

- **Use a property of quantifiers**
 - For all x there exists y = There is no x such that there is no y

$$\forall X \exists Y = \neg \exists X \neg \exists Y$$

Division in TRC with Double Negation

- Names of coffeehouses that sell all coffees

- All coffeehouses B such that there is no coffee E such that there is no tuple in Sells for B and E

$$\{ R \mid \exists H \in \text{Coffeehouses} \forall C \in \text{Coffees} \exists S \in \text{Sells} ($$

$$H.\text{name} = S.\text{coffeehouse} \wedge$$

$$C.\text{name} = S.\text{coffee} \wedge$$

$$R.\text{name} = H.\text{name}) \}$$

$$\{ R \mid \exists H \in \text{Coffeehouses} \neg \exists C \in \text{Coffees} \neg \exists S \in \text{Sells} ($$

$$H.\text{name} = S.\text{coffeehouse} \wedge$$

$$C.\text{name} = S.\text{coffee} \wedge$$

$$R.\text{name} = H.\text{name}) \}$$

Division in TRC with Double Negation

Translated into SQL

- Names of coffeehouses that sell all coffees
 - All coffeehouses B such that there is no coffee E such that there is no tuple in Sells for B and E

```
{ R |  $\exists H \in \text{Coffeehouses} \neg\exists C \in \text{Coffees} \neg\exists S \in \text{Sells} ($   
    H.name = S.coffeehouse  $\wedge$   
    C.name = S.coffee  $\wedge$   
    R.name = H.name ) }
```

```
sql  
  
select H.name  
from Coffeehouses H  
where not exists (  
    select *  
    from Coffees C  
    where not exists (  
        select *  
        from Sells S  
        where H.name = S.coffeehouse  
        and C.name = S.coffee));
```

Example Query

- Names of drinkers who frequent all coffeehouses

sql

```
select D.name
from Drinkers D
where not exists (
    select *
    from Coffeehouses H
    where not exists (
        select *
        from Frequents F
        where D.name = F.drinker
        and H.name = F.coffeehouse
    )
);
```

Practice

- **Names of coffees that are sold at all coffeehouses**

Double Negation vs. Counting

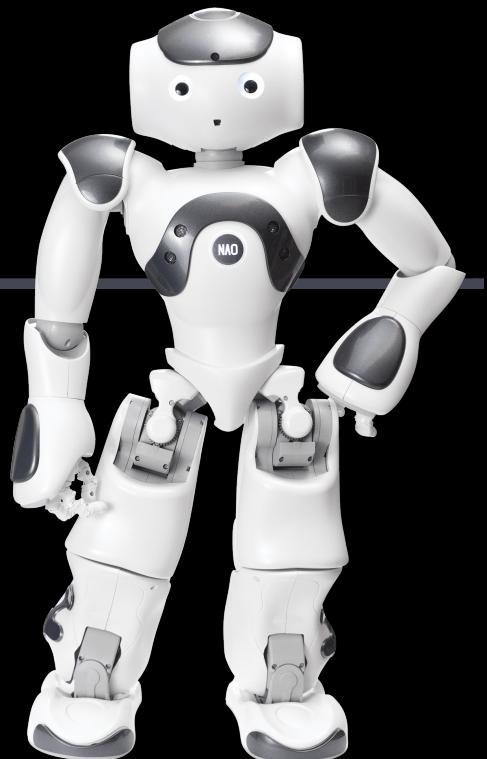
- Are they completely equivalent?

sql

```
select C.name  
from Coffees C  
where not exists (  
    select *  
    from Coffeehouses H  
    where not exists (  
        select *  
        from Sells S  
        where H.name = S.coffeehouse  
        and C.name = S.coffee));
```

csharp

```
select S.coffee  
from Sells S  
group by S.coffee  
having count(coffeehouse) = (  
    select count(*)  
    from Coffeehouses );
```



The Operator ANY

- **$x = \text{ANY}(<\text{subquery}>)$ is a boolean condition that is true iff x equals at least one tuple in the subquery result.**
 - $=$ could be any comparison operator.
- **Example: $x > \text{ANY}(<\text{subquery}>)$ means x is not the uniquely smallest tuple produced by the subquery (greater than at least one).**
 - Note tuples must have one component only.

sql

```
SELECT coffee
FROM Sells
WHERE price > ANY (
    SELECT price
    FROM Sells );
```

Practice

- For each coffeehouse, show all coffees that are more expensive than some other coffee sold at that bar

The Operator ALL

- $x \diamond \text{ALL}(<\text{subquery}>)$ is true iff for every tuple t in the relation, x is not equal to t .
 - That is, x is not in the subquery result.
- \diamond can be any comparison operator.
- Example: $x \geq \text{ALL}(<\text{subquery}>)$ means there is no tuple larger than x in the subquery result.
- From Sells(coffeehouse, coffee, price), find the coffee(s) sold for the highest price.

```
sql
SELECT coffee
FROM Sells
WHERE price >= ALL(
    SELECT price
    FROM Sells
    WHERE price IS NOT NULL);
```

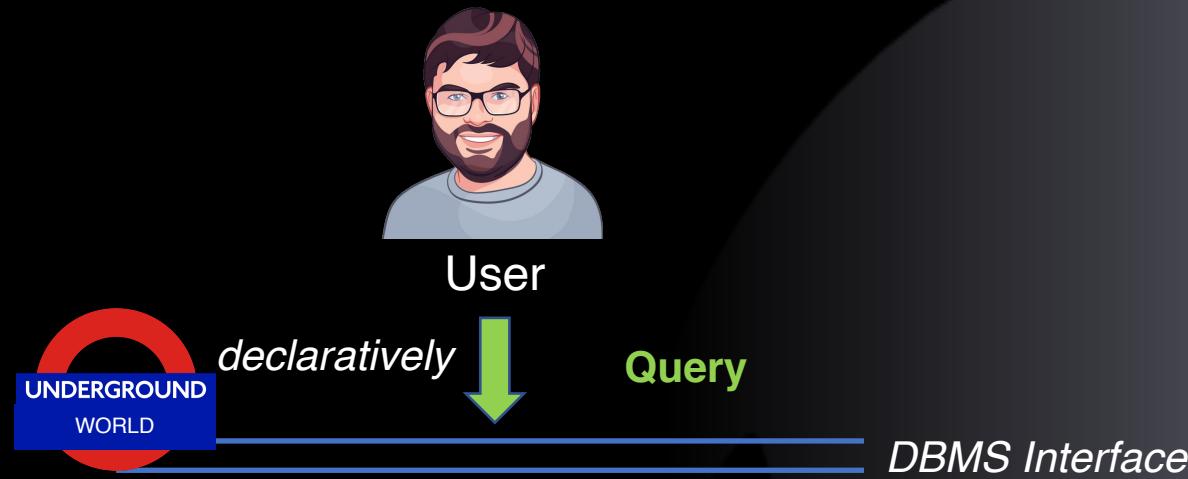
Practice

- **For each coffeehouse, show the most expensive coffee(s) sold at that coffeehouse**

Subqueries in FROM

- A parenthesized SELECT-FROM-WHERE statement (subquery) can be used as a value in a number of places, including WHERE and FROM clauses.
- In place of a relation in the FROM clause, we can use a subquery and then query its result.
- Must use a tuple-variable to name tuples of the result.
- Find the coffees liked by at least one person who frequents Joe's.

```
sql
SELECT DISTINCT coffee
FROM Likes
JOIN (
    SELECT drinker
    FROM Frequents
    WHERE coffeehouse = 'Joe''s'
) JD ON Likes.drinker = JD.drinker;
```



SQL DML -- Views

Readings:

PDBM 1

What is a View?

- **A virtual table constructed from actual tables on the fly**
 - Can be accessed in queries like any other table
 - Not materialized, constructed when accessed
 - Similar to a subroutine in ordinary programming
 - Is a schema element

```
CREATE VIEW Name(Columns, ... )  
AS  
SQL QUERY
```

Example from the Sports DB

- For each result, the name of the athlete, the name of the sport, and the percentage of the record achieved by the result (a result that is a record should therefore appear as 100; this column should be named “percentage”).

```
vbnet
SELECT P.name, S.name, ROUND(100*(R.result/S.record),0) AS 'percentage'
FROM People P, Results R, Sports S
WHERE P.ID = R.peopleID
AND S.ID = R.sportID;
```

```
vbnet
CREATE VIEW E2Q10 (PID, pname, SID, sname, percentage)
AS
SELECT P.ID, P.name, S.ID, S.name, ROUND(100*(R.result/S.record),0)
FROM People P
JOIN Results R ON P.ID = R.peopleID
JOIN Sports S ON S.ID = R.sportID;
```

```
sql
SELECT pname, sname, percentage
FROM E2Q10;
```

Practice from the Sports DB

- Create a view for this query and write the query using the view: The ID, name and gender of all athletes who participated in the competition held in Hvide Sande in 2009.

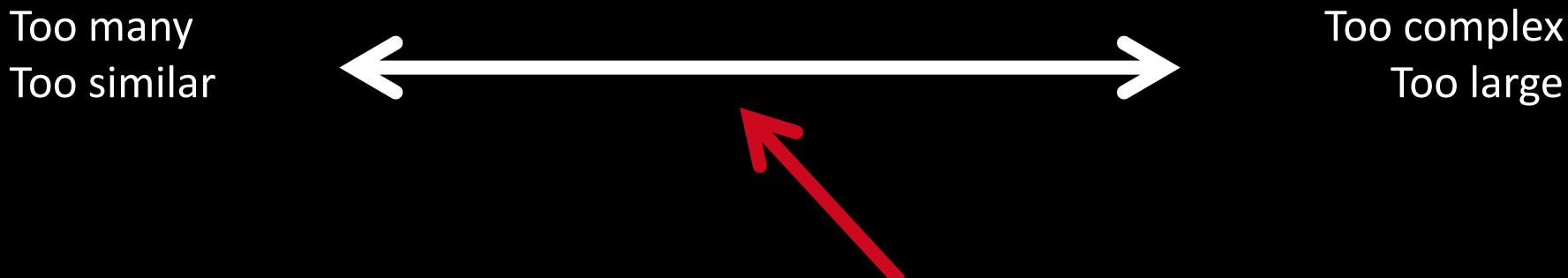
```
sql Copy code
SELECT DISTINCT P.ID, P.name, G.description
FROM People P
JOIN Gender G ON P.gender = G.gender
JOIN Results R ON P.ID = R.peopleID
JOIN Competitions C ON C.ID = R.competitionID
WHERE C.place = 'Hvide Sande' AND EXTRACT(YEAR FROM C.held) = 2009;
```

- Write the query using the view for: The name and gender of all people with last name that starts with a “J” and ends with a “sen”.

```
vbnet
SELECT P.name, G.description
FROM People P
JOIN Gender G ON P.gender = G.gender
WHERE P.name LIKE '% J%sen';
```

Why Views?

- Simpler queries – for development, maintenance!
- Code re-use – view used across applications
- Access management – grant access to views only
- Physical data independence – views can mask changes
- Can be overused!



Takeaways

- **1 Block**
 - select, from, where, group by, having, sort by clauses
- **Many query blocks**
 - Multi-block queries
 - Connected by UNION, EXCEPT, INTERSECT
 - Sub-queries in FROM or WHERE clause
 - Blocks connected by =, IN, EXISTS, ANY, ALL
 - Opposites (usually) with NOT
 - BUT: Try to use JOIN when possible!
 - Division is one important type of queries with >1 query block!



What is next?

- **Next week: Triggers, Functions, Transactions, SQL & DBMS programming in Java**
 - One single lecture for both B.Sc. and M.Sc.
- **Homework 1 is out**
 - learnIT: instructions and quiz (homework)
 - Help from Tas today at 14:15
 - Deadline: 24.02.2023 at 23:59:59
 - Remember: 3 out of 4 MAs to sit on the exam

chatGPT

Introduction to Database Systems

I2DBS – Spring 2023

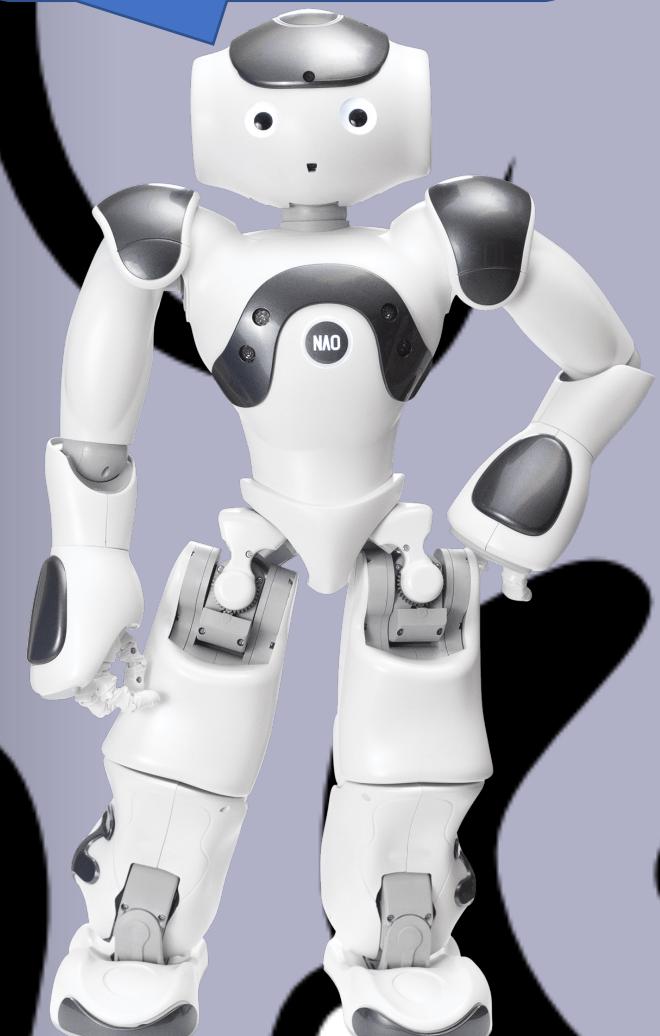
- Week 4:
- SQL Programming
- DBMS Programming: JDBC
- Transactions

Jorge-Arnulfo Quiané-Ruiz

Readings:

PDBM 9.2, 14.1, 14.2.1, 14.5

Hello, let's go on!



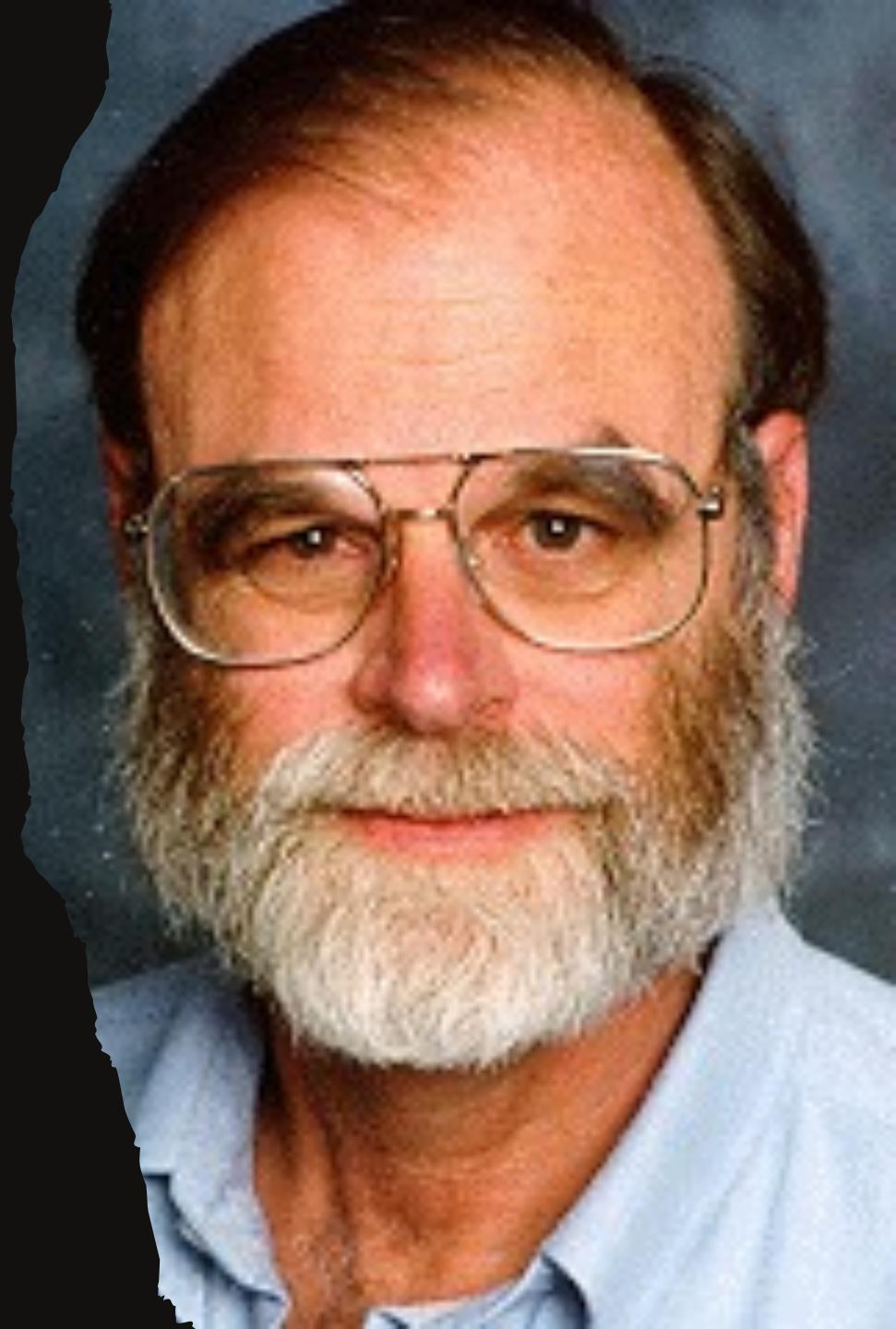


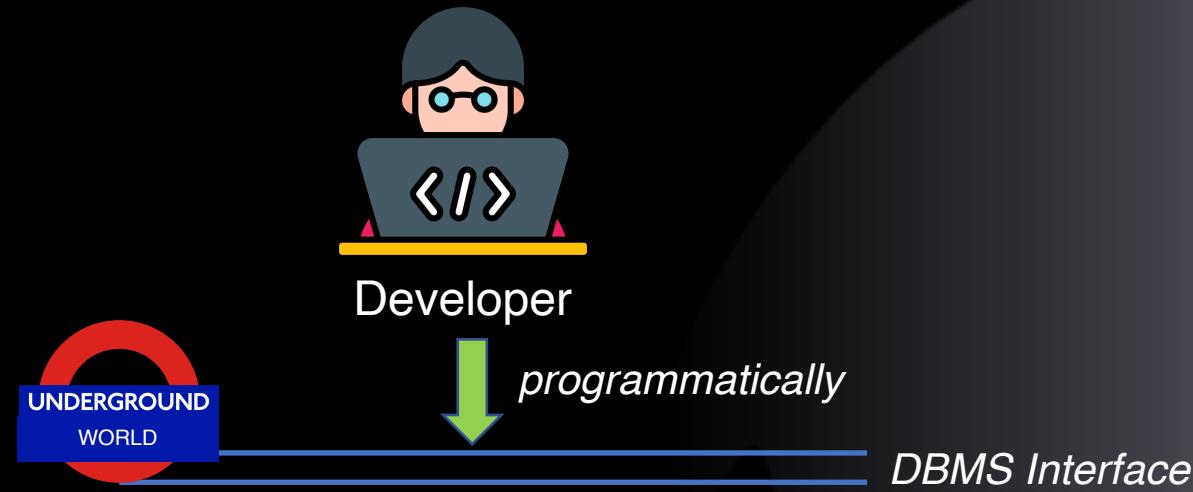
Profile of the Week

James Nicholas Gray (Jim Gray)

Father of Transaction Processing

- **1944:** Born in San Francisco, California
- **1969:** PhD in Computer Science from UC, Berkeley
- **1970--2007:** Joined IBM, Tandem Computers, and MSR
- **Mid-70s:** Co-developed System R relational
- **1998:** Received the Turing Award
- **2007:** disappeared while sailing off the coast of California





SQL Programming

Database Functions

- A database trigger is a function that is automatically executed in response to certain events on a particular table (or view) in a database
 - Events are typically INSERT, UPDATE, DELETE
 - Triggers can execute BEFORE or AFTER the event
- Triggers can be used for maintaining the integrity of the information on the database.
 - For example, when a new record (representing a new worker) is added to the employees table, new records could also be created in the tables of taxes, vacations and salaries.
- Triggers can also be used to log historical data, for example to keep track of employees' previous salaries.

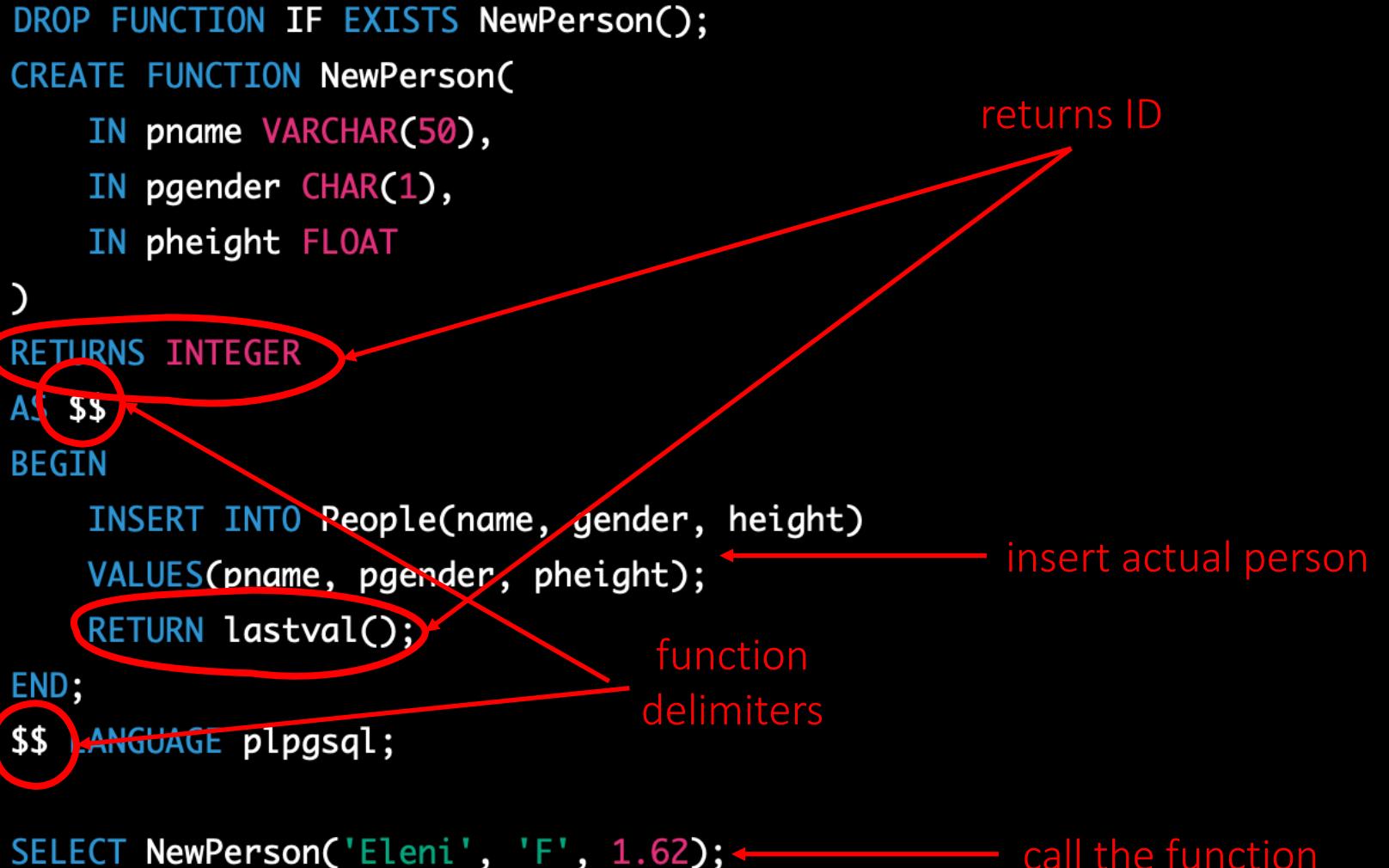
Create New Person

sql

 Copy code

```
DROP FUNCTION IF EXISTS NewPerson();
CREATE FUNCTION NewPerson(
    IN pname VARCHAR(50),
    IN pgender CHAR(1),
    IN pheight FLOAT
)
RETURNS INTEGER
AS $$
BEGIN
    INSERT INTO People(name, gender, height)
    VALUES(pname, pgender, pheight);           ← insert actual person
    RETURN lastval();                         ← function
                                              delimiters
END;
$$ LANGUAGE plpgsql;

SELECT NewPerson('Eleni', 'F', 1.62);        ← call the function
```



The annotations are as follows:

- A red circle highlights "RETURNS INTEGER". A red arrow points from this circle to the text "returns ID" located above the function body.
- A red circle highlights "AS \$\$". A red arrow points from this circle to the text "function delimiters" located below the RETURN statement.
- A red circle highlights "\$\$ LANGUAGE plpgsql;". A red arrow points from this circle to the text "call the function" located at the bottom of the code block.
- A red circle highlights "VALUES(pname, pgender, pheight);". A red arrow points from this circle to the text "insert actual person" located next to the INSERT statement.

Using Functions

- Typically from Java via JDBC or other framework
- Can also be done from an SQL script:

```
sql
-- Query 1
SELECT NewPerson('Alice', 'F', 1.62);

-- Query 2
SELECT * FROM NewPerson('Alice', 'F', 1.62);

-- Query 3
DO $$

    BEGIN
        PERFORM NewPerson('Alice', 'F', 1.62);
    END
$$;
```

Example: Find Biggest Increase in Record (I)

- **Function BiggestRecordJump**
 - Input: ID of sport
 - Output: The largest increase of that sport
- **Assume RecordLog table with old and new**
 - Later: Maintain this table automatically!

sql

```
CREATE TABLE RecordLog (
    peopleID INT,
    competitionID INT,
    sportID INT,
    oldrecord FLOAT,
    newrecord FLOAT,
    seton DATE
);
```

Example: Find Biggest Increase in Record (II)

sql

```
DROP FUNCTION IF EXISTS BiggestRecordJump();

CREATE FUNCTION BiggestRecordJump(IN sid INT)
RETURNS FLOAT
AS $$

DECLARE
    r FLOAT;
BEGIN
    SELECT MAX(newrecord - oldrecord) INTO r
    FROM RecordLog
    WHERE sportID = sid;
    RETURN r;
END $$ LANGUAGE plpgsql;
```

 Copy code

sql

```
SELECT BiggestRecordJump(1); -- replace 1 with the desired sportID value
```

Are Functions Faster?

- **May be faster than executing the code from a client**
Why?
- **Code may be pre-compiled and optimized**
 - Do not need to invoke optimizer again
 - But this may happen with well written queries using plan caching
- **The code runs at the server**
 - The server may be more efficient
 - No need to move data to client

Pros and Cons of Functions

Pros

- **Code shared across ALL applications**
- **May be used for access control**
- **May give performance benefits**

Cons

- **Very system-specific**
- **Code maintenance requires care**
 - Versioning is difficult

Database Triggers

- **May be faster than executing the code from a client**

Why?

- **Code may be pre-compiled and optimized**

- Do not need to invoke optimizer again
 - But this may happen with well written queries using plan caching

- **The code runs at the server**

- The server may be more efficient
 - No need to move data to client

Triggers in Postgres

- **Multiple triggers per table per event**
 - INSERT / UPDATE / DELETE
 - Run in alphabetical order
 - Per row OR Per operation – focus here on per row
- **New data is in the NEW record**
 - For UPDATE / INSERT
 - Same schema as modified relation
 - Can refer to NEW.ID, NEW.name, ...
- **Old data is in the OLD record**
 - For UPDATE / DELETE
 - Same schema, refer to OLD.ID, OLD.name, ...
- **The variable TG_OP says which operation it is**

Example 1: Check Values

Test Code

```
insert into Results values (1, 1, 1, -1.0);
```

check new value

error handling

when

operations

```
sql  
  
DROP TRIGGER IF EXISTS CheckResult ON Results;  
DROP FUNCTION IF EXISTS CheckResult();  
  
CREATE FUNCTION CheckResult()  
RETURNS TRIGGER  
AS $$ BEGIN  
    IF(NEW.result < 0.0) THEN  
        RAISE EXCEPTION 'CheckResult: Result must be positive!'  
        USING ERRCODE = '45000';  
    END IF;  
    RETURN NEW;  
END; $$ LANGUAGE plpgsql;  
  
-- Actually create the trigger, only for INSERT or UPDATE  
CREATE TRIGGER CheckResult  
BEFORE INSERT OR UPDATE  
ON Results  
FOR EACH ROW EXECUTE PROCEDURE CheckResult();
```

Example 2: Ban Updates/Deletes

throw raise exception

```
sql Copy code

-- Drop trigger if it already exists
DROP TRIGGER IF EXISTS BanChanges ON Results;

-- Drop function if it already exists
DROP FUNCTION IF EXISTS BanChanges();

-- Create the BanChanges function
CREATE FUNCTION BanChanges()
RETURNS TRIGGER
AS $$
BEGIN
    RAISE EXCEPTION 'BanChanges: Cannot change results!';
    USING ERRCODE = '45000';
END;
$$ LANGUAGE plpgsql;

-- Create the BanChanges trigger for UPDATE or DELETE
CREATE TRIGGER BanChanges
BEFORE UPDATE OR DELETE
ON Results
FOR EACH ROW
EXECUTE PROCEDURE BanChanges();

-- Test the trigger by attempting to delete a row from Results
DELETE FROM Results
WHERE sportID = 3;
```

Example 3: Update Record

find matching record in Sports
update sport record

sql

```
-- Actually create the trigger, only for INSERT or UPDATE
CREATE TRIGGER UpdateRecord
AFTER INSERT OR UPDATE
ON Results
FOR EACH ROW EXECUTE PROCEDURE UpdateRecord();

-- Test it
insert into Results values (1, 1, 1, 100000.0);
select * from Sports where ID = 1;
```

sql

```
-- Drop trigger if it already exists
DROP TRIGGER IF EXISTS UpdateRecord ON Results;

-- Drop function if it already exists
DROP FUNCTION IF EXISTS UpdateRecord();

-- Create the UpdateRecord function
CREATE FUNCTION UpdateRecord()
RETURNS TRIGGER
AS $$
BEGIN
    IF NEW.result > (
        SELECT S.record
        FROM Sports S
        WHERE S.ID = NEW.sportID) THEN
        UPDATE Sports
        SET record = NEW.result
        WHERE ID = NEW.sportID;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Example 4: Log Changes (I)

- Keep track of all new records in a RecordLog

sql

 Copy code

```
DROP TABLE IF EXISTS RecordLog;

CREATE TABLE RecordLog (
    sportID INT,
    peopleID INT,
    competitionID INT,
    oldrecord FLOAT,
    newrecord FLOAT,
    seton DATE,
    PRIMARY KEY (peopleID, competitionID, sportID),
    FOREIGN KEY (peopleID, competitionID, sportID)
        REFERENCES Results (peopleID, competitionID, sportID)
);
```

Example 4: Log Changes (II)

sql

Copy code

```
DROP TRIGGER IF EXISTS LogRecord ON Results;  
DROP FUNCTION IF EXISTS LogRecord();
```

```
CREATE FUNCTION LogRecord()  
RETURNS TRIGGER  
AS $$  
DECLARE  
    oldRecord FLOAT;  
BEGIN  
    IF NEW.result > (  
        SELECT S.record  
        FROM Sports S  
        WHERE S.ID = NEW.sportID) THEN  
        SELECT S.record INTO oldRecord  
        FROM Sports S  
        WHERE S.ID = NEW.sportID;  
        INSERT INTO RecordLog  
        VALUES (NEW.sportID, NEW.peopleID, NEW.competitionID,  
                oldRecord, NEW.result, CURRENT_DATE);  
    END IF;  
    RETURN NEW;  
END; $$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER LogRecord  
AFTER INSERT ON Results  
FOR EACH ROW EXECUTE PROCEDURE
```

sql

```
-- Test it  
INSERT INTO Results VALUES (1, 1, 2, 100000.0);  
SELECT * FROM RecordLog;
```

Questions

- Why not create the LogRecord trigger on Sports?
- What about trigger order?
 - LogRecord vs UpdateRecord
 - Order is alphabetical = good in this case
 - Try it with a different order!
 - May be better to merge similar triggers?
- What about BEFORE or AFTER?
 - LogRecord and UpdateRecord are AFTER
 - CheckResult and BanUpdates are BEFORE
 - BEFORE and AFTER what? When does each apply? Why?

Example 5: Merged Trigger

sql

Copy code

```
CREATE FUNCTION MergedTrigger()
RETURNS TRIGGER
AS $$ BEGIN
    IF (TG_OP = 'DELETE' OR TG_OP = 'UPDATE') THEN
        RAISE EXCEPTION 'MergedTrigger: Cannot change results!'
        USING ERRCODE = '45000';
    END IF;
    IF( NEW.result < 0.0) THEN
        RAISE EXCEPTION 'MergedTrigger: Result must be positive!'
        USING ERRCODE = '45000';
    END IF;
    RETURN NEW;
END; $$ LANGUAGE plpgsql;

-- Actually create the trigger
CREATE TRIGGER MergedTrigger
BEFORE INSERT OR UPDATE OR DELETE
ON Results
FOR EACH ROW EXECUTE PROCEDURE MergedTrigger();

-- Test it
INSERT INTO Results VALUES (1, 1, 3, -1.0);
DELETE FROM Results WHERE sportID = 3;
```

BEFORE vs. AFTER triggers

- Are you only checking the newly inserted/updated entry? → BEFORE (or AFTER)
 - Checking happens earlier with BEFORE, so less work
 - But if other triggers might modify the values, then prefer AFTER
- Are you inserting a row to another table with a foreign key constraint to the NEW record? → AFTER
 - Otherwise, the NEW record is NOT in the database, so your insertion will fail!
- Are you modifying the NEW record? → BEFORE
 - Otherwise, the record is already in the database and will not be changed
- Are you doing both 2 and 3? → BEFORE and AFTER
 - Two different triggers!

Pros and Cons of Database Code

Pros

- **Code often runs faster**
 - No context switch
 - Compiled code
 - No data transfers
- **Useful for security**
 - Wrap data and functionality
 - Same access from all clients
- **Same code for all applications**

Cons

- **Code is “hidden”**
 - Only visible via system tables
 - Easily forgotten
 - Versioning is hard!
 - Schema may be edited using a GUI = no-no!
- **Generally not portable**

Exercises on SQL and DBMS Programming

- **Exercises 4 are out**

- Views, procedures, triggers
- Use test script to verify your work
 - Can run parts in pgAdmin – best via command prompt
 - You can extend the script with your own tests!
- Also has a very nice database cleaning script

```
# once
psql -q Bills < bills-schema.sql
```

```
#repeatedly
psql Bills < universal-cleanup.sql
psql Bills < your-solution.sql
psql Bills < test-script.sql > output 2>&1
less output
```

Catalog

- All information about the database is stored in tables!

- This is the catalog!

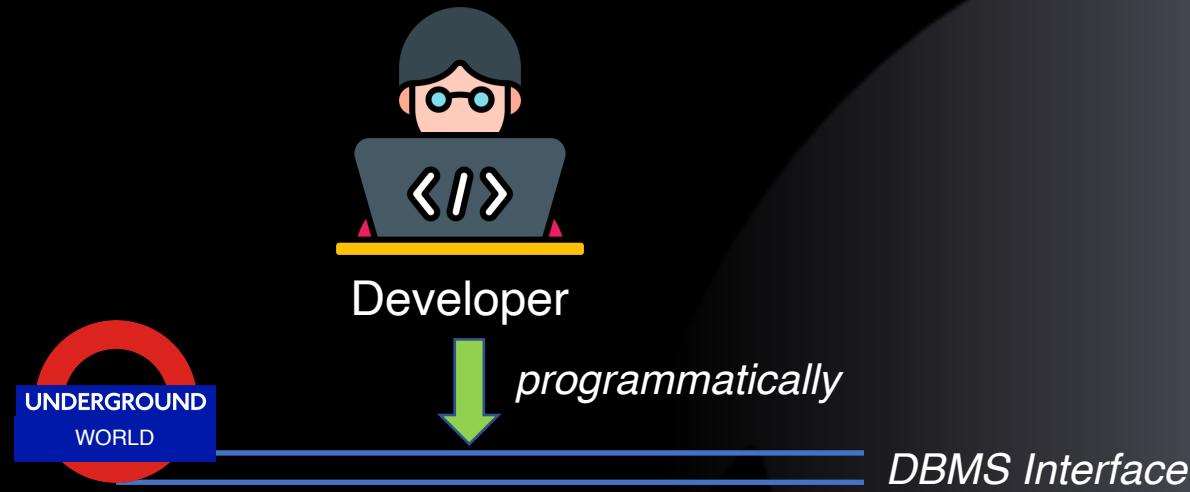
```
select *  
from information_schema.tables  
where table_schema = 'public';
```

<https://www.postgresql.org/docs/current/catalogs.html>

- See also: System information functions

<https://www.postgresql.org/docs/current/functions-info.html>

```
select current_database();  
select current_user;  
select lastval();
```

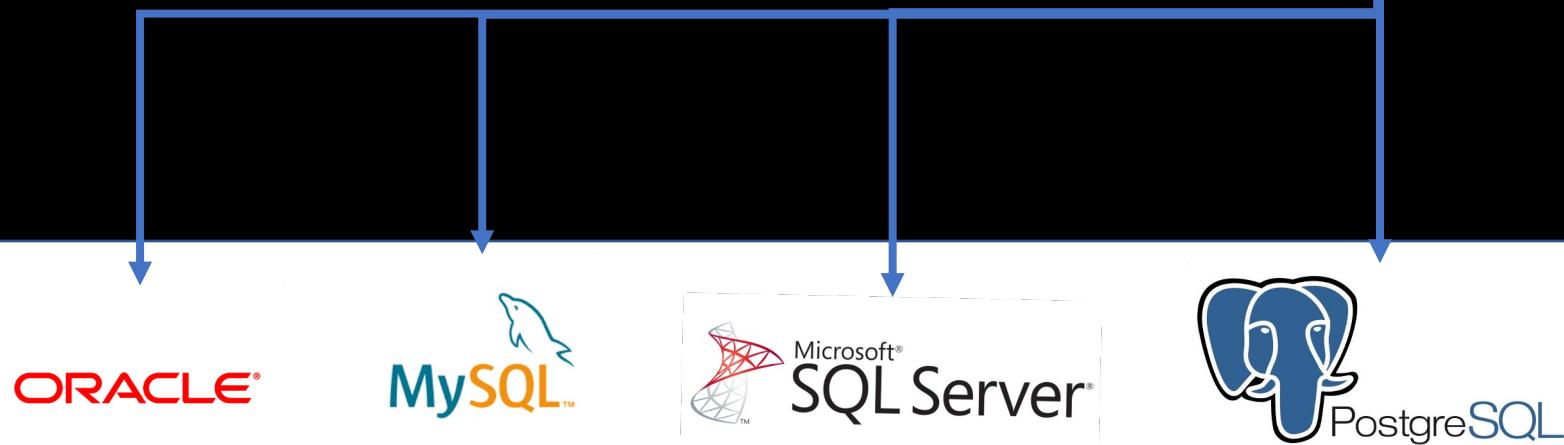


DBMS Programming

Problem: Vendor Lock-In

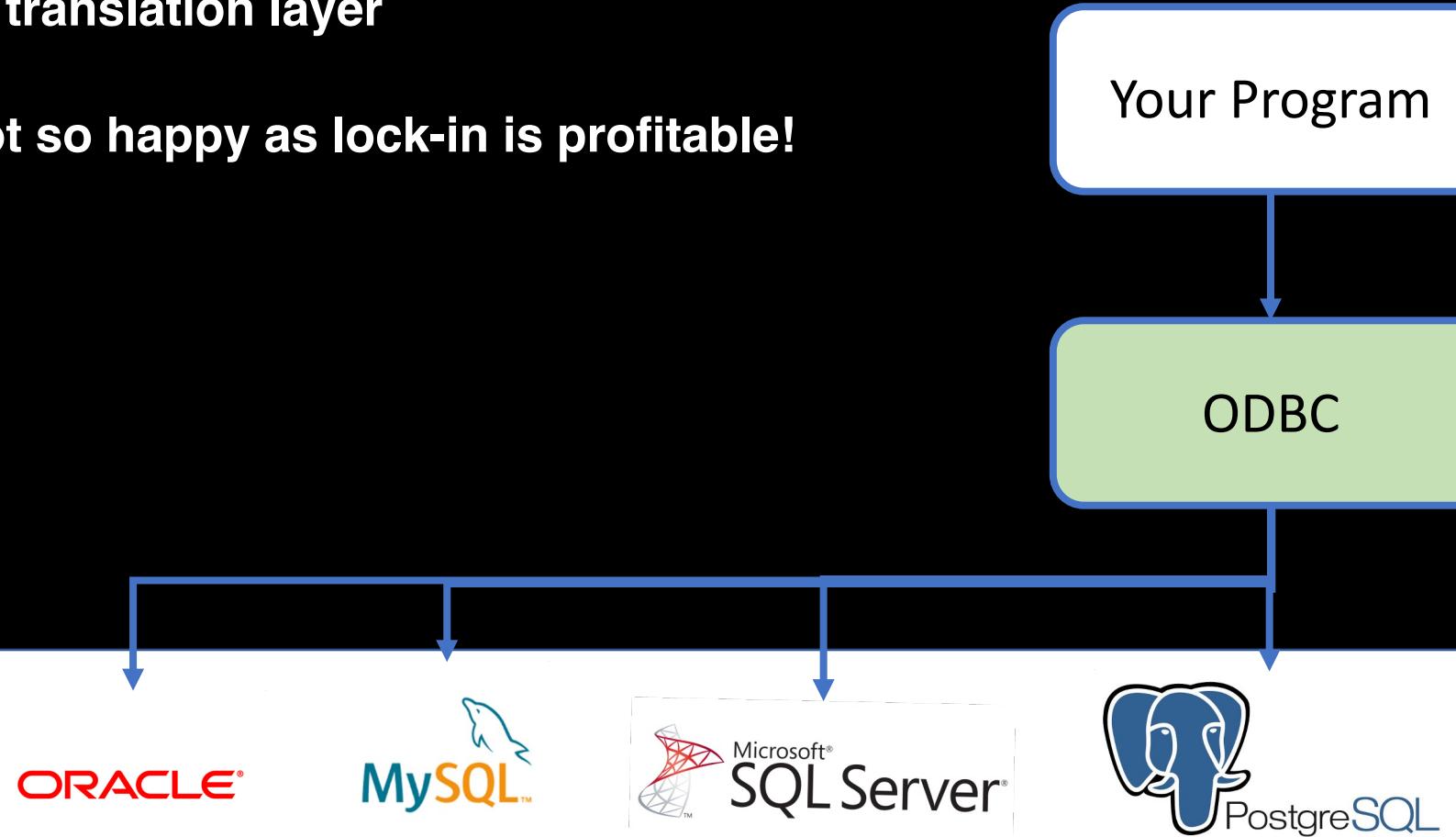
- They're all compatible with standard SQL...
- and mutually incompatible due to vendor-specific functionality!

Your Program



Open Database Connectivity (ODBC)

- In-between translation layer
- Vendors not so happy as lock-in is profitable!

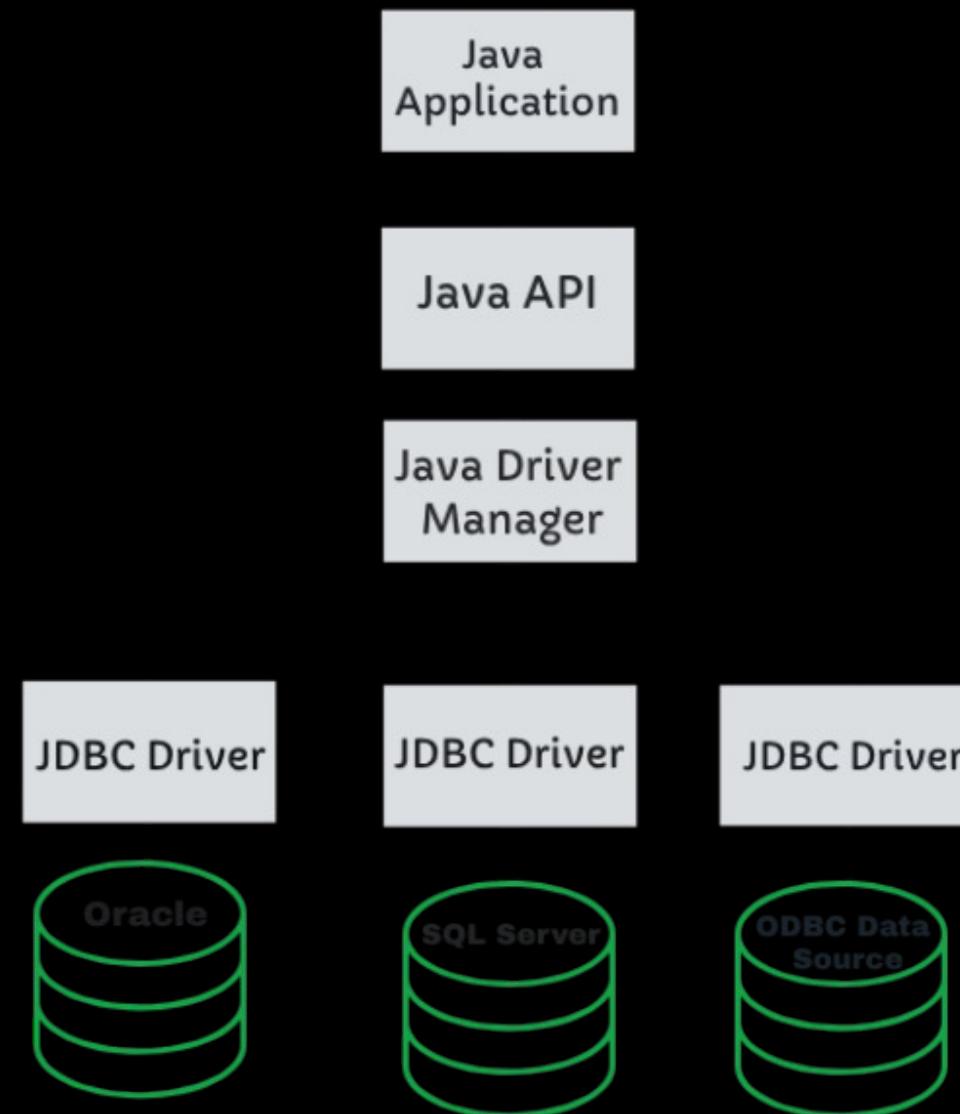


Java Database Connectivity (JDBC)

- A database API provided by Java
- JDBC is included in the Java Development Kit (JDK)
- JDBC provides a standard set of interfaces and classes

Java Database Connectivity (JDBC)

JDBC Architecture



JDBC API



Connection

java

 Copy code

```
String connectionString = "jdbc:postgresql://localhost:5432/myDB";
Connection conn = DriverManager.getConnection(
    connectionString, user, password);
// run some statements on the connection
conn.close();
```

credentials

database path

Statement

number of affected records

success or failure

```
java

// assume that Connection conn is already connected
Statement st = conn.createStatement();
int count = st.executeUpdate("INSERT INTO product (name,price) VALUES
    ('Coffee', '10.00')");
boolean returnType = st.execute("SELECT * FROM product");
st.close();
```

Statement: Multiple ResultSets

The diagram illustrates the execution of a single SQL statement that returns multiple ResultSets. A red line connects the first `getResultSet()` call to the label "first ResultSet". Another red line connects the second `getResultSet()` call to the label "second ResultSet".

```
scss
// assume that Connection conn is already connected
Statement st = conn.createStatement();

st.execute("SELECT COUNT(*) FROM person; SELECT * FROM person");
ResultSet rs = st.getResultSet();
if (rs.next()) System.out.println("Count: " + rs.getString(1));

st.getMoreResults();
ResultSet rs2 = st.getResultSet();
while (rs2.next()) System.out.println("Name: " + rs2.getString(1));
```

Copy code

Statement: Example

scss

```
// assume that Connection conn is already connected
Statement st = conn.createStatement();
st.executeUpdate("DELETE * FROM person");
st.close();
```

Statement: Parameter Injection

variable: value can change

variable injection

```
java Copy code
int personId = 5;
Statement st = conn.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM person WHERE id=" + personId);
// do something with the ResultSet...
st.close();
```

Statement: Code Refactoring (Functions)

function definition

function call

```
java // Disclaimer: this code is not great, it is only a pedagogical tool  
ResultSet getPersonById(int personId) {  
    Statement st = conn.createStatement();  
    return st.executeQuery("SELECT * FROM person WHERE id=" + personId);  
}  
  
ResultSet rs = getPersonById(1);  
while (rs.next()) {  
    System.out.println("name: " + rs.getString("name"));  
}  
rs.close();
```

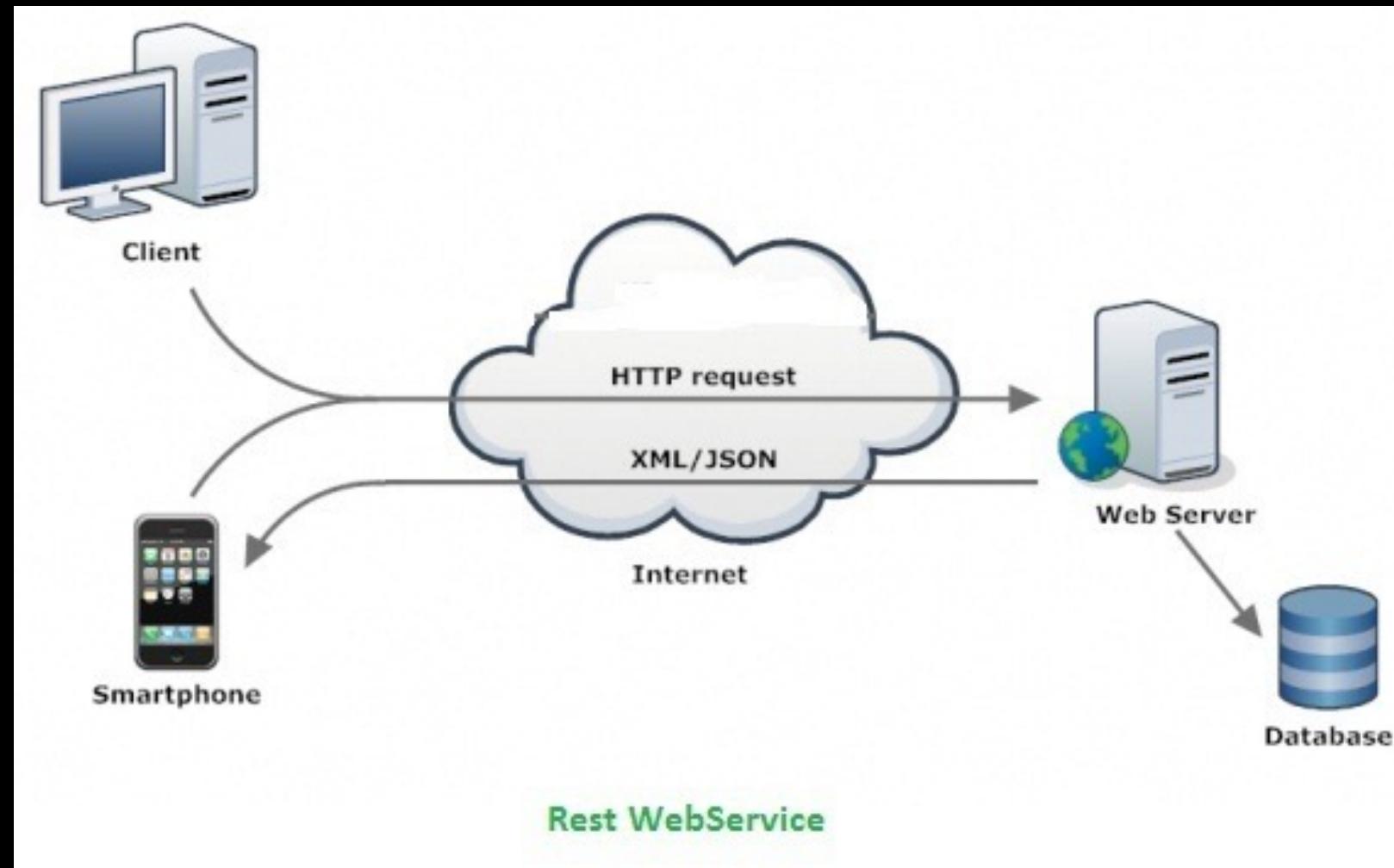
Copy code

ResultSet (Part 1)

iterating over the results

```
java Copy code
int salary = 5000;
Statement st = conn.createStatement();
ResultSet rs = st.executeQuery(
    "SELECT * FROM person WHERE salary > " + salary);
while (rs.next()) {
    System.out.println("name: " + rs.getString("name"));
}
rs.close();
st.close();
```

WebService Example



Statement: SQL Injection



javascript

Copy code

```
ResultSet getPersonByName(String name) {
    Statement st = conn.createStatement();
    return st.executeQuery("SELECT * FROM person WHERE name='" + name + "'");
}

// Webservice URL: http://company.com/getPersonByName?name=Bob
// ... does the following:
ResultSet rs = getPersonByName("Bob");
while (rs.next()) {
    System.out.println("name: " + rs.getString("name"));
}
rs.close();
```

Statement: SQL Injection - An Attack Example



original query

injected query

```
scss
ResultSet getPersonByName(String name) {
    Statement st = conn.createStatement();
    return st.executeQuery(
        "SELECT * FROM person WHERE name='"
        + name + "'");
}

// Webservice URL:
// http://company.com/getPersonByName?name='
// <-- DELETE FROM TABLE person WHERE ' 1 ' = ' 1 '
// ... does the following:
ResultSet rs = getPersonByName(
    "; DELETE FROM TABLE person WHERE ' 1 ' = ' 1 ''");
while (rs.next()) {
    System.out.println("name: " + rs.getString("name"));
}

// Query that is called:
// "SELECT * FROM person WHERE name= ';
// DELETE FROM TABLE person WHERE ' 1 ' = ' 1 ' '"
```

PreparedStatement

values setting

query execution

java

parameterized query

```
PreparedStatement st = conn.prepareStatement(  
    "SELECT * FROM person WHERE name=? AND balance>?");
```

// First call

```
st.setString(1, "Alice");
```

```
st.setInt(2, 10);
```

```
ResultSet rs = st.executeQuery();
```

```
while (rs.next()) {
```

```
    System.out.println("Name: " + rs.getString("name"));
```

```
}
```

// Second call

```
st.setString(1, "Brian");
```

```
st.setInt(2, 20);
```

```
ResultSet rs2 = st.executeQuery();
```

```
while (rs2.next()) {
```

```
    System.out.println("Name: " + rs2.getString("name"));
```

```
}
```

ResultSet (Part 2)

if too large, fetched in blocks

if not closed, it continues
taking resources

```
java

Statement st = conn.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM person");
while (rs.next()) {
    // ...
}

// Close the statement to release server resources
st.close();
```

SQL Exception: Catch & Propagate

you catch and handle with it

you propagate the exception

java

```
// Exception propagation
void example2() throws SQLException {
    // statement.execute(sql)
}
```

csharp

```
// Using try-catch
void example1() {
    try {
        // statement.execute(sql)
    } catch (SQLException e) {
        System.out.println("An error occurred");
        return;
    }
}
```

SQL Exception: Try-Finally & Try-With

closed whatever it happens

closed automatically

vbnet

```
// Using try-finally (releases resources correctly)
Statement st = conn.createStatement();
try {
    st.execute("this is invalid sql");
} catch (Exception e) {
    throw e;
} finally {
    st.close();
}
```

vbnet

```
// Using try-with (releases resources correctly)
try (Statement st = conn.createStatement()) {
    System.out.println("testTryFinally: try");
    st.execute("this is invalid sql");
}
```

SQL Query Timeout

- timeout=0 *disable timeout*
- timeout=30 *normal queries (30 seconds)*
- timeout=600 *large / complex queries (10 minutes)*

java

 Copy code

```
// Create a Statement object
Statement statement = connection.createStatement();

// Set the query timeout value to 30 seconds
statement.setQueryTimeout(30);

// Execute a query
ResultSet resultSet = statement.executeQuery("SELECT * FROM my_table");
```



RDBMS

Transactions

A Simple Example

Step 1: Get balance of Janet
SELECT Balance ...

AccountName	Balance
Janet	200
Alice	0

Step 2: Reduce balance of Janet
UPDATE accounts SET Balance=150
WHERE Name='Janet'

AccountName	Balance
Janet	150
Alice	0

- Task: Transfer 50 DKK from Janet to Alice

Step 3: Get balance of Alice
SELECT Balance ...

AccountName	Balance
Janet	150
Alice	0

Step 4: Increase balance of Alice
UPDATE accounts SET Balance=50
WHERE Name='Alice'

AccountName	Balance
Janet	150
Alice	50

What Is a Transaction?

- **A group of operations to the database**
 - Wish for “all or nothing” execution
 - Wish for isolation from other transactions
 - Ex: Transfer = Withdrawal + Deposit

- **Basic SQL transactions**

BEGIN;
COMMIT;
ROLLBACK;

- **Savepoints**
 - SAVEPOINT <name>;
 - ROLLBACK TO SAVEPOINT <name>;

More About Postgres Transactions

- **By default, every statement is a transaction**
 - To override this behaviour:
 - BEGIN; ... COMMIT; / ROLLBACK;
 - Some (DDL) statements implicitly COMMIT transactions
- **Calling a function starts a transaction**
 - Can assume that the function has transactional properties!
 - Errors abort the transaction, erase all previous operations!
- **Errors inside functions:**
 - Cannot simply say: ROLLBACK
 - Need to raise and handle exceptions!
 - See examples in code...

Transactions Properties

- **A**tomicity
 - Transaction is “one operation”
- **C**onsistency
 - Each transaction moves the DB from one consistent state to another
- **I**solation
 - Each transaction is alone in the world
- **D**urability
 - Each transaction moves the DB from one consistent state to another

Constraints
Triggers

Concurrency
Control

Recovery

Transaction Implementation Methods

- **Consistency**
 - Keys and foreign keys = limited DBMS support
- **Isolation**
 - Historically locking = strict two-phase locking
 - Recently multi-version concurrency control
- **Atomicity / Durability**
 - Logging all changes to disk
 - Write ahead logging protocol

Transactions and Testing

- One use for transactions is in *test scripts*
 - Many examples in Exercise 4
- • Simple pattern to test changing the database without actually changing the database

```
BEGIN
  -- Make changes
  -- Run test queries
ROLLBACK
```

Transactions in JDBC

- Task: Transfer 50 DKK from Janet to Alice
(not using JDBC auto-commit)

lua

 Copy code

```
st.execute("BEGIN; " +
    "UPDATE accounts SET Balance=Balance - 50 WHERE Name='Janet'; "
    "UPDATE accounts SET Balance=Balance + 50 WHERE Name='Alice'; "
    "COMMIT;");
```

JDBC Auto Commit

- Task: Transfer 50 DKK from Janet to Alice

lua

 Copy code

```
st.execute("UPDATE accounts SET Balance=Balance - 50 WHERE Name='Janet'");  
st.execute("UPDATE accounts SET Balance=Balance + 50 WHERE Name='Alice'");
```



lua

 Copy code

```
st.execute("BEGIN; " +  
          "UPDATE accounts SET Balance=Balance - 50 WHERE Name='Janet'; "  
          "COMMIT; " +  
          "BEGIN; " +  
          "UPDATE accounts SET Balance=Balance + 50 WHERE Name='Alice'; "  
          "COMMIT;");
```

Disabling JDBC Auto Commit

- Task: Transfer 50 DKK from Janet to Alice

scss

 Copy code

```
conn.setAutoCommit(false);

// Still inserts BEGIN before the statement
st.execute("UPDATE accounts SET Balance=Balance - 50 WHERE Name='Janet'");
st.execute("UPDATE accounts SET Balance=Balance + 50 WHERE Name='Alice'");

conn.commit(); // Executes "COMMIT"
conn.setAutoCommit(true); // If you forgot this line, you messed up!!!
```

A JDBC Template for a Safe Transaction

- Task: Transfer 50 DKK from Janet to Alice

java

 Copy code

```
try (Statement st = conn.createStatement()) {  
    conn.setAutoCommit(false);  
    st.execute("UPDATE accounts SET Balance=Balance - 50 " +  
              "WHERE Name='Janet' ");  
    st.execute("UPDATE accounts SET Balance=Balance + 50 " +  
              "WHERE Name='Alice' ");  
    conn.commit();  
} catch (Exception e) {  
    conn.rollback();  
} finally {  
    conn.setAutoCommit(true);  
}
```

Takeaways

- 
- **Functions**
 - Set of database operations, performance benefits
 - **Triggers**
 - Execute a function in response to a database
 - **JDBC**
 - Java API for database operations
 - **Transactions**
 - Atomicity, Consistency, Isolation, Durability



What is next?

- **Next week: Exercise 4M**
 - no lecture = recap study week
- **Next Lecture:**
 - ER-Diagrams
 - Translation to SQL DDL

Introduction to Database Systems

I2DBS – Spring 2023

- Week 5:
- ER Diagrams
- Translation to SQL DDL

Jorge-Arnulfo Quiané-Ruiz

Readings:

PDBM 3.0-3.3, 6.3-6.4

Information

- **Homework: We will provide feedback ASAP**
 - Solution will be uploaded on LearnIT
 - Also review document on LearnIT
 - common errors for selected queries
- **Exercise 5: ER design and implementation**
 - It is large and comprehensive
 - More than 2 hrs, but use it to practice...
- **Homework 2 opens on Friday**
 - Try to finish pending exercises first



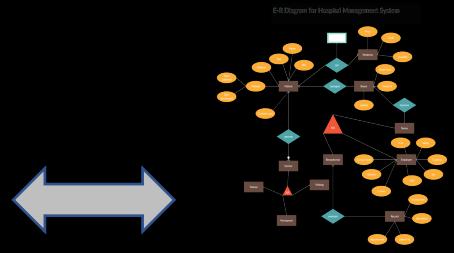
Profile of the Week

Peter Chen

Inventor of the Entity-Relationships Model

- **1947:** Born in Taichung, Taiwan
- **1973:** PhD in Computer Science from Harvard University
- **1974-1978:** Assist. Prof. at MIT Sloan School of Management
- **1976:** Published the ER Model
- **1978-1983:** Assoc. Prof. at UCLA
- **1983-2011:** Distinguished Prof. at LSU





Modelling
(ER Diagram)

Entity-Relationships Model

Readings:

PDBM 3.0-3.3, 6.3-6.4

Why Conceptual Model?

- The “boss” knows she wants a database...
... but not what should be in it!
- Need an effective method to develop the schema and document its structure

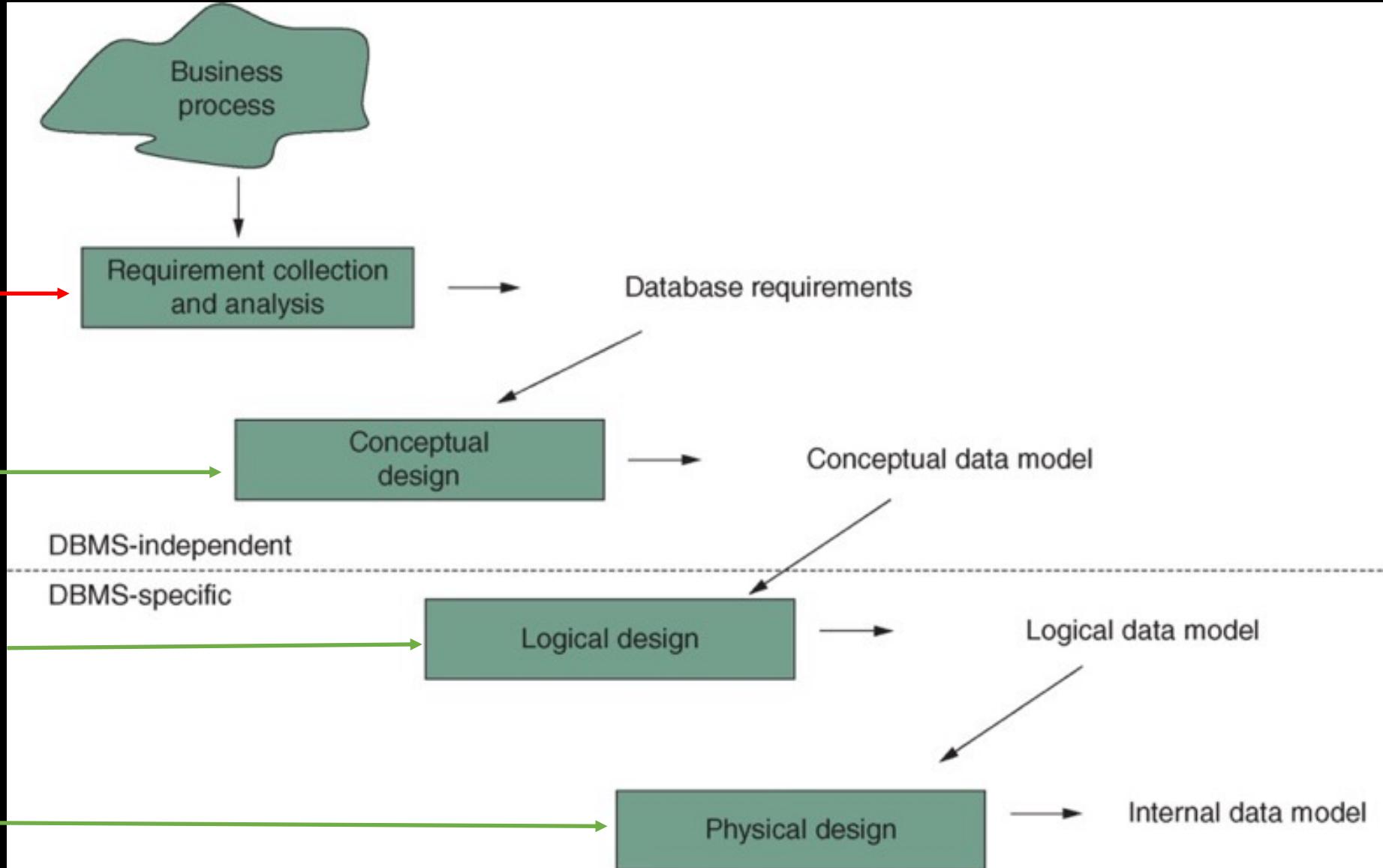
Design Process

Not in this course

Sketch the key components
in an ER-diagram

Convert ER-diagram to DDL

Weeks 7 & 8



ER: Entity - Relationships

- **Conceptual Model Defined by Peter Chen (1976)**
- **ER = modeling concepts + visual representation**
 - ER/EER notation is not standardized
 - Every textbook/company/tool has its own visual representation
 - Core concepts are universally accepted
 - EER vs ER rarely distinguished → we just call it ER!
- **UML can be used as design notation**
 - Slight differences – no UML in this course
- **Focus on ER notation from the book**
 - ... plus, some minor extensions – made clear in the lectures
 - In exercises, project, exam: Use the notation in book, lectures

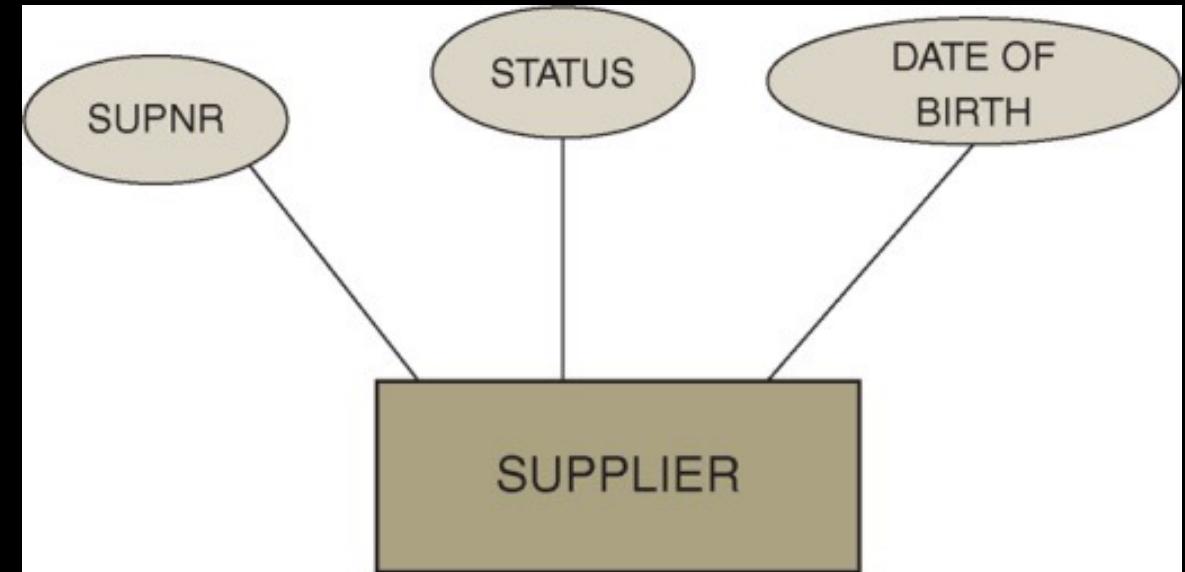
Entity Types and Attribute Types

- **Entity Type**

- Set of similar “things”
- Ex: students, courses
- Entity = instance
- Ex: John, CSE305

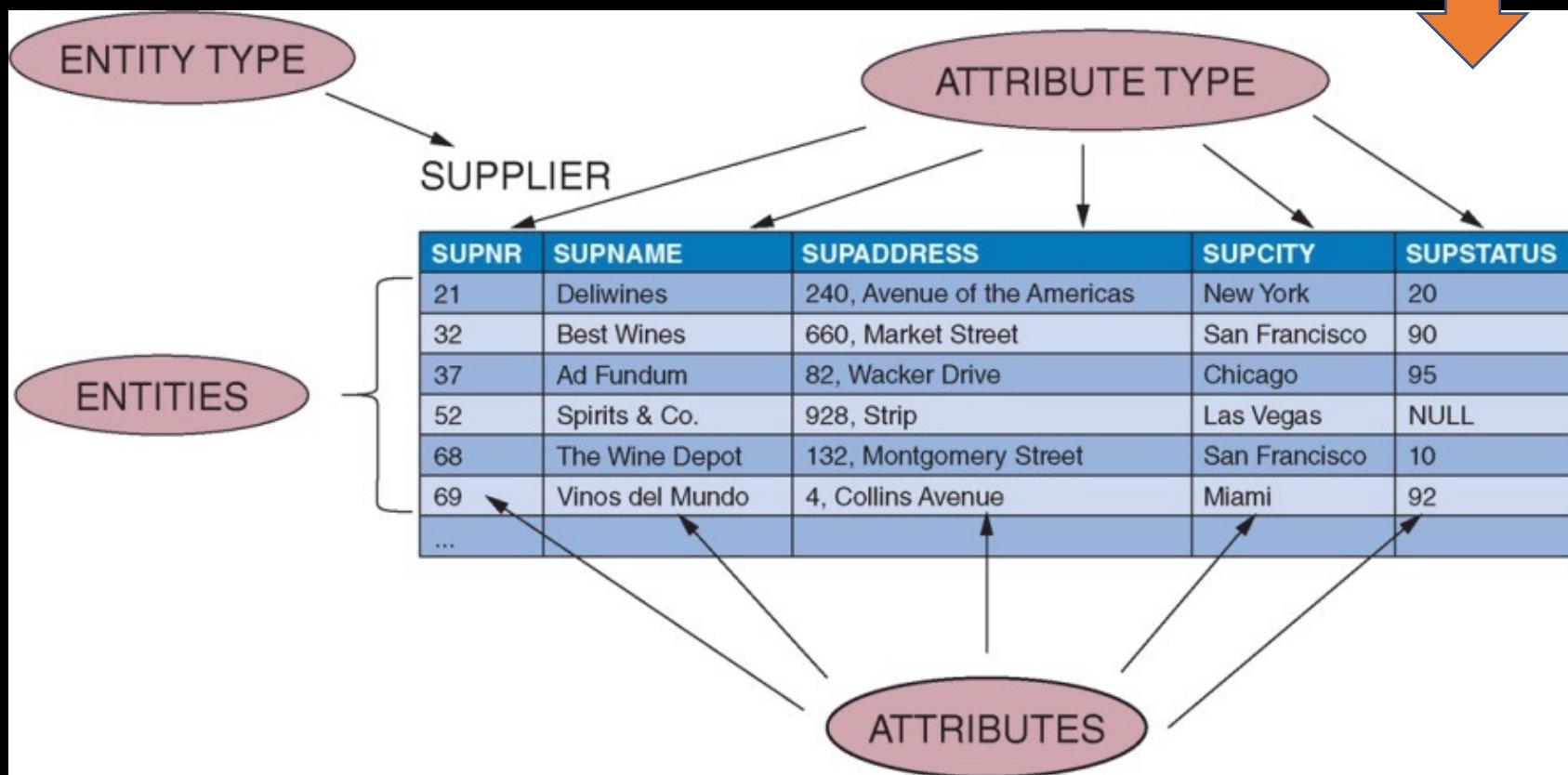
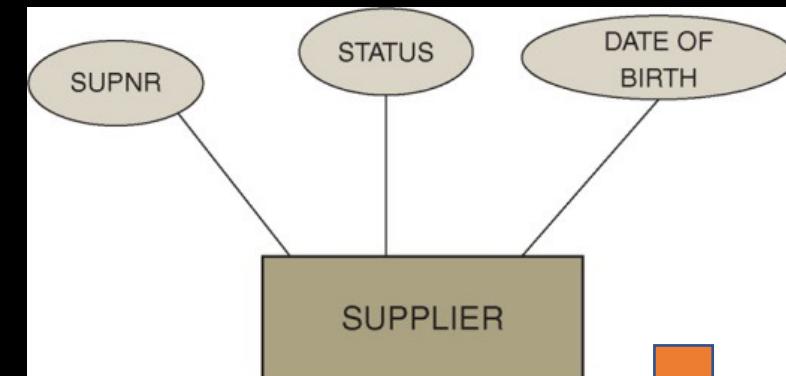
- **Attribute Type**

- Describes one aspect of an entity set
- Ex: name, maximum enrolment

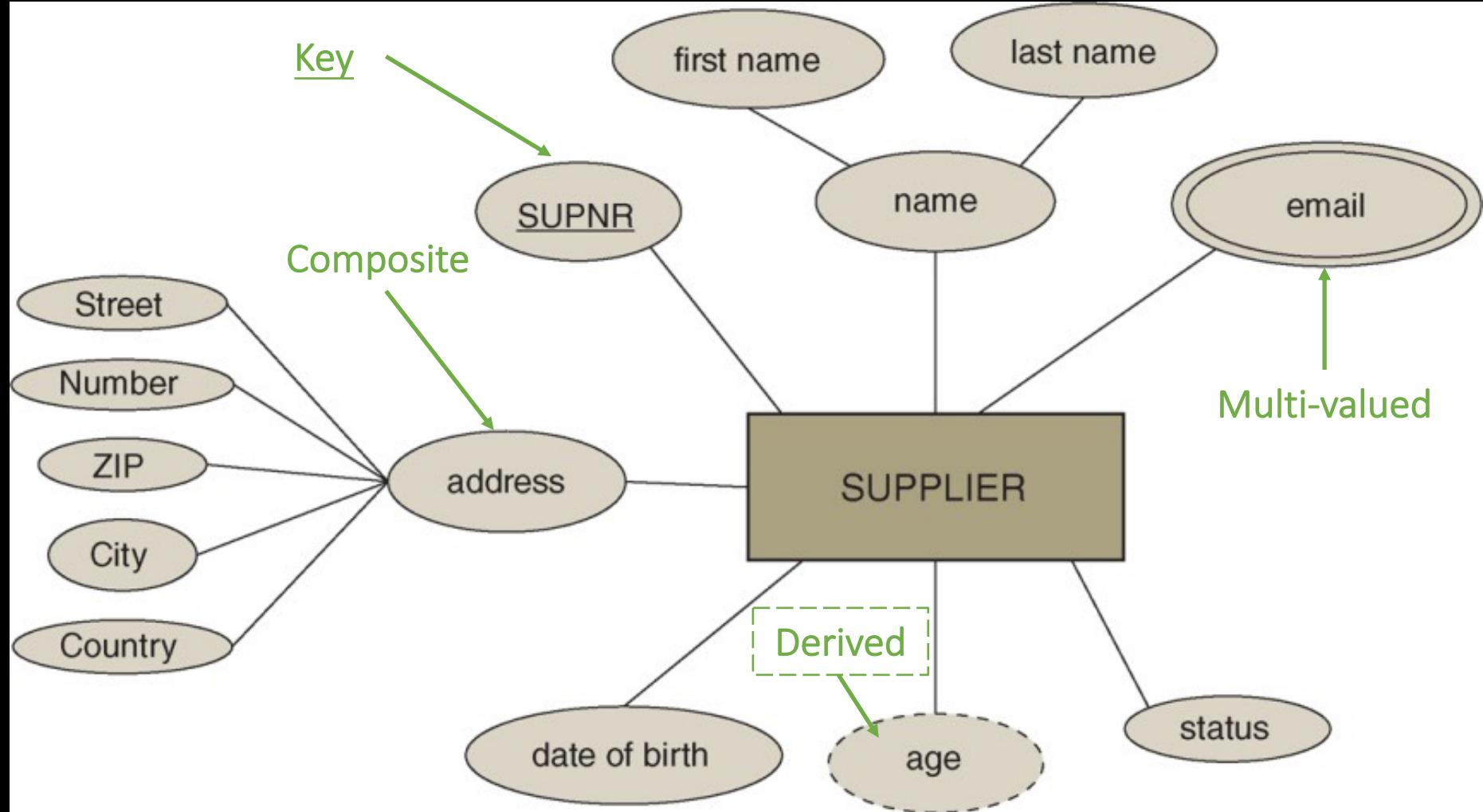


Entity Type as a Relation

```
CREATE TABLE Supplier (
    SUPNR INT PRIMARY KEY,
    SUPNAME VARCHAR NOT NULL,
    ...
);
```



More Attribute Types



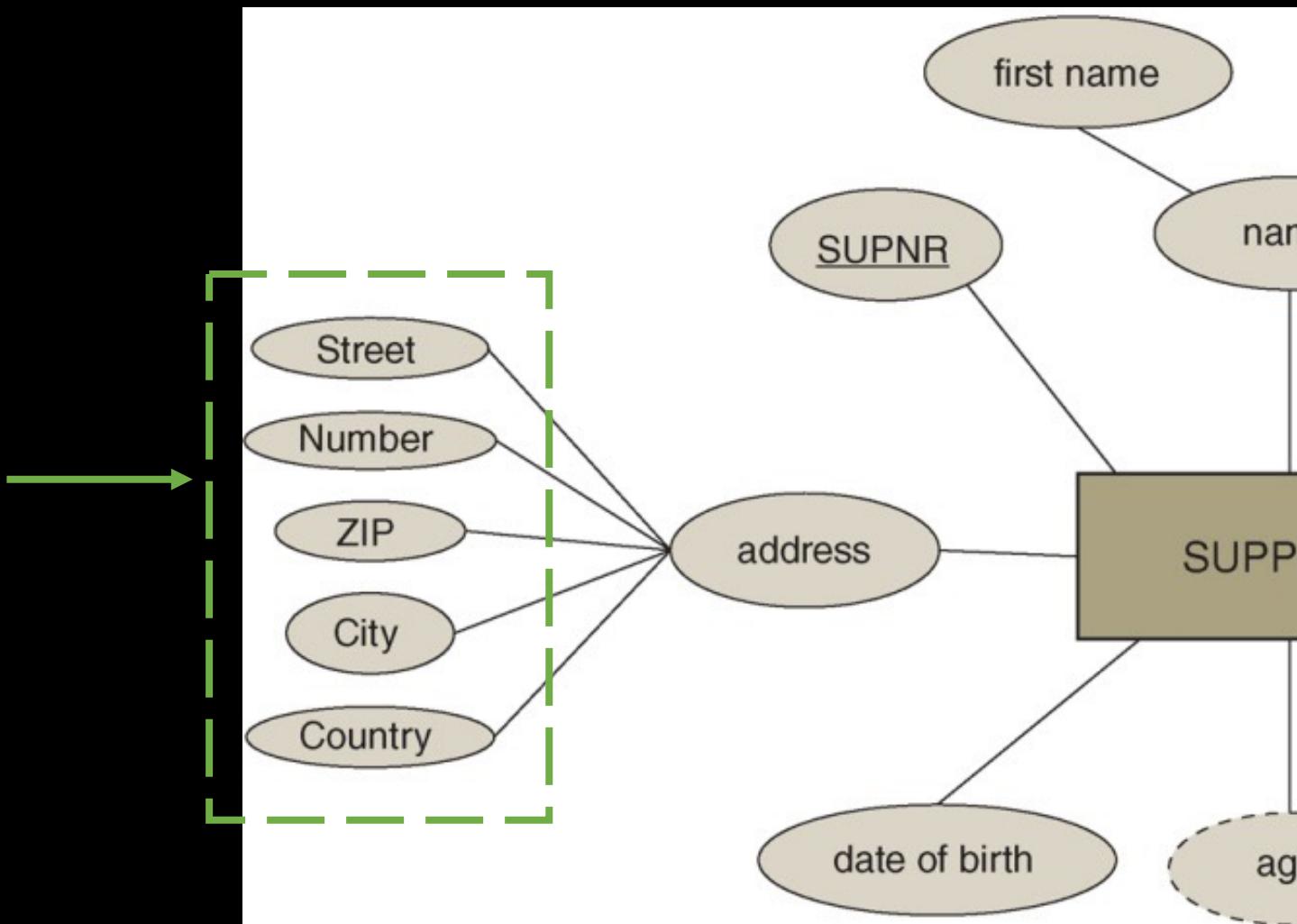
Keys in ER Diagrams and SQL DDL

- **Underlined key = PRIMARY KEY**
- **ER diagrams cannot show secondary keys**
 - They must be noted somewhere else!
 - They must still be UNIQUE in the SQL table!

Composite Attributes in SQL DDL

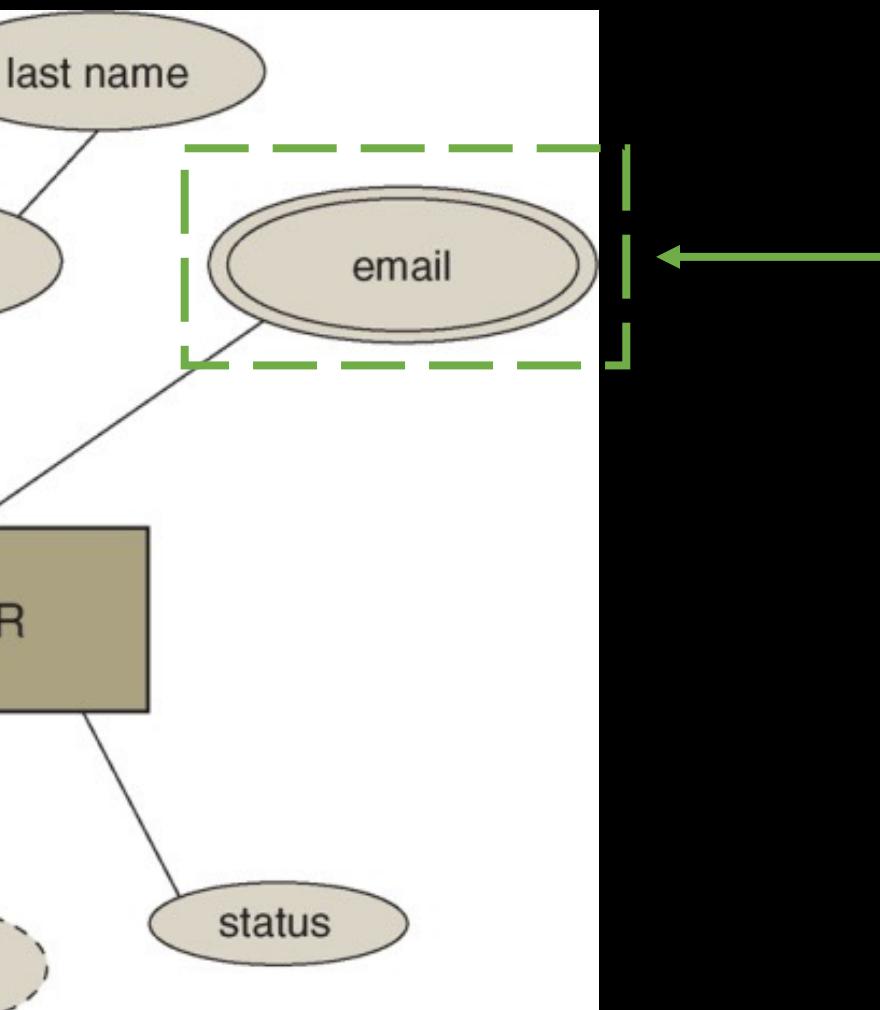
- Simply use the detailed attributes

```
CREATE TABLE Supplier (
    ...
    Street VARCHAR NOT NULL,
    Number INTEGER NOT NULL,
    ZIP INTEGER NOT NULL,
    City VARCHAR NOT NULL,
    Country VARCHAR NOT NULL,
    ...
);
```



Multivalued Attributes in SQL DDL

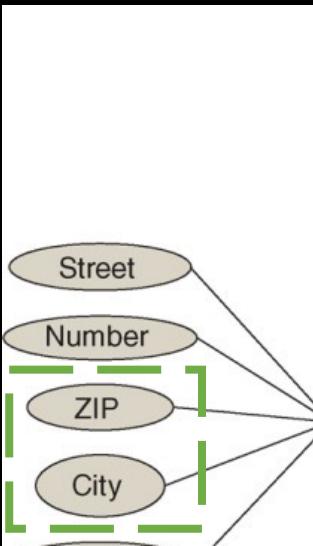
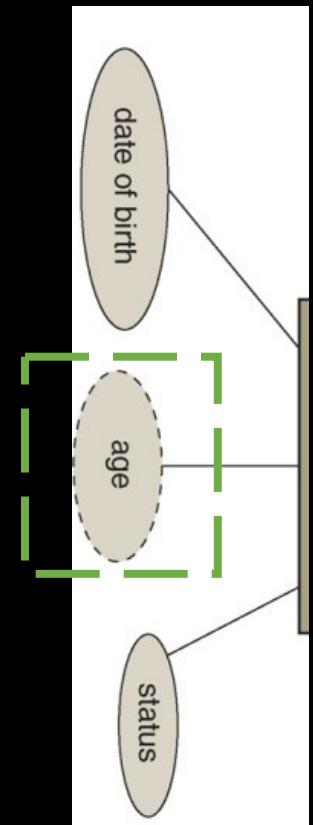
- Create a new table referencing the entity table



```
CREATE TABLE Emails (
    Email VARCHAR,
    SupNR INTEGER
    REFERENCES Supplier(SupNR),
    PRIMARY KEY (Email, SupNR)
);
```

Derived Attributes in SQL DDL

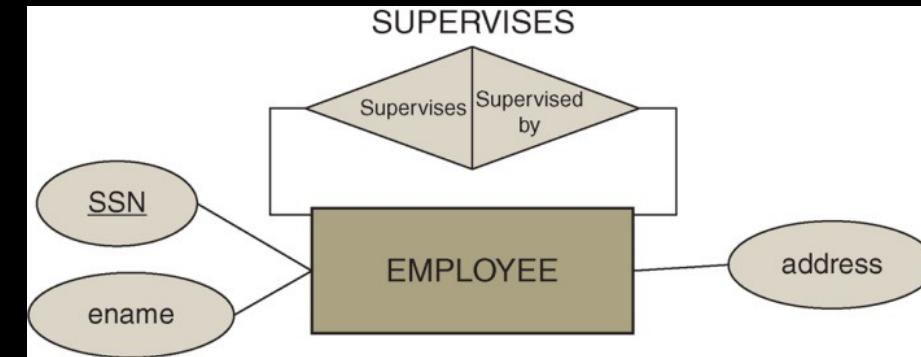
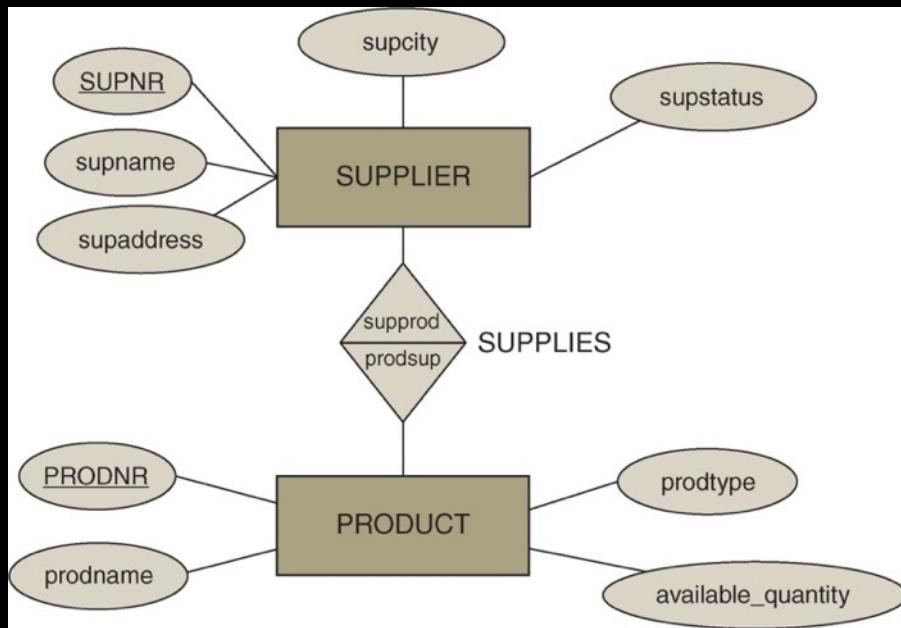
- Not discussed in the PDBM book!
- Option 1: Create an attribute and maintain it
 - E.g. with a trigger, or regular update processes
- Option 2: Create a view that computes it
- Neither is very good!
- Sometimes there is a second kind of inter-attribute relationship
 - Here: ZIP → City
 - Called functional dependencies (FDs)
 - ER diagrams may miss such relationships!
 - We fix this with normalization (week 6)



Relationship Types

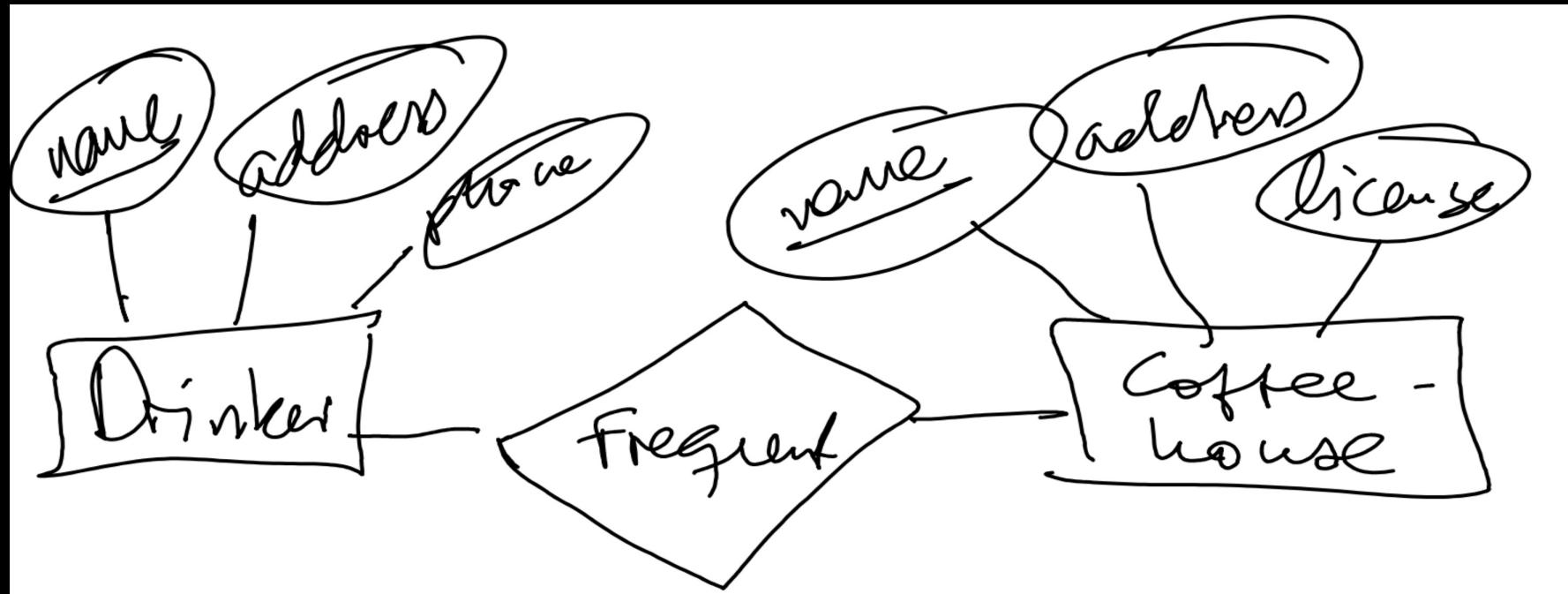
- **Relationship Types**

- Relate two or more entities (with roles)
- Ex: John majors in Computer Science
- Roles may be omitted when obvious



Exercise

- For drinker, store (unique) name, address, and phone.
- For coffeehouses, store (unique) name, address, and license.
- Store which drinkers frequent which coffeehouses.

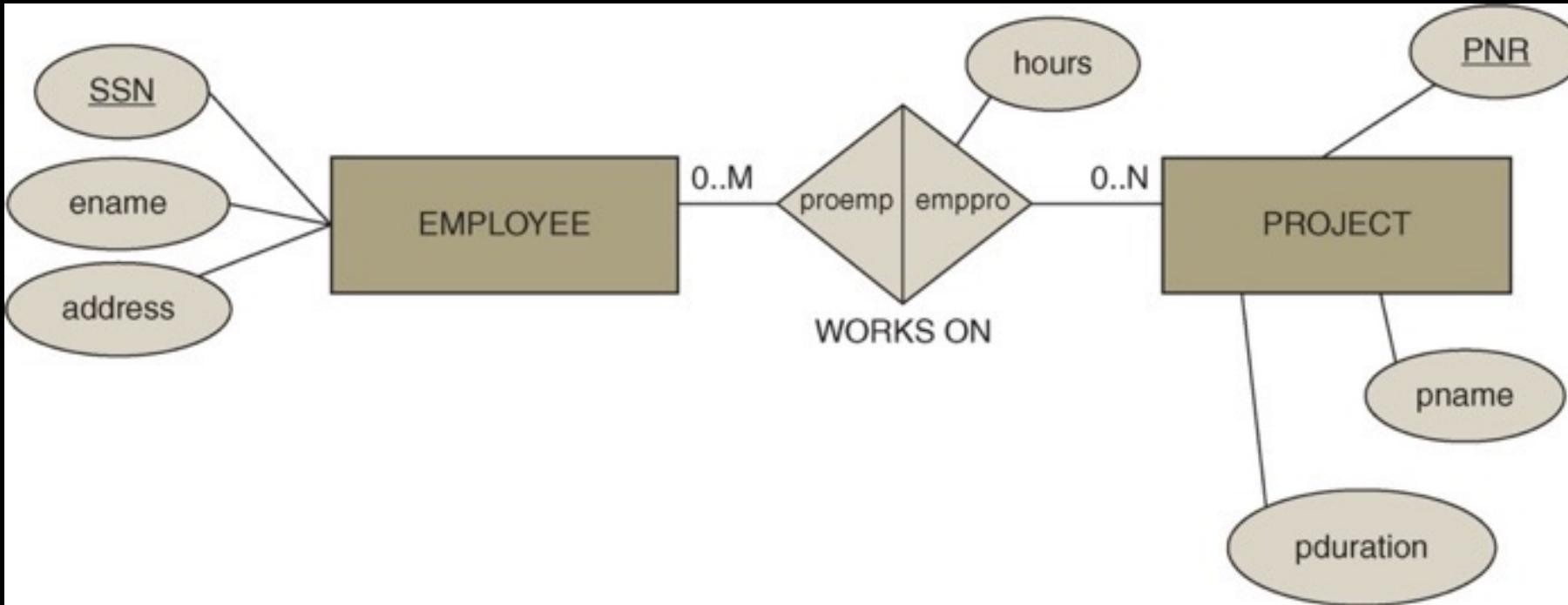


Relation vs. Relationship Type

- **Relation (relational model)**
 - set of tuples
- **Relationship type (ER model)**
 - describes relationship between entities of an enterprise
- **Both entity types and relationship types (from an ER model) will be represented by relations (in the relational model)**

Relationship Attribute Types

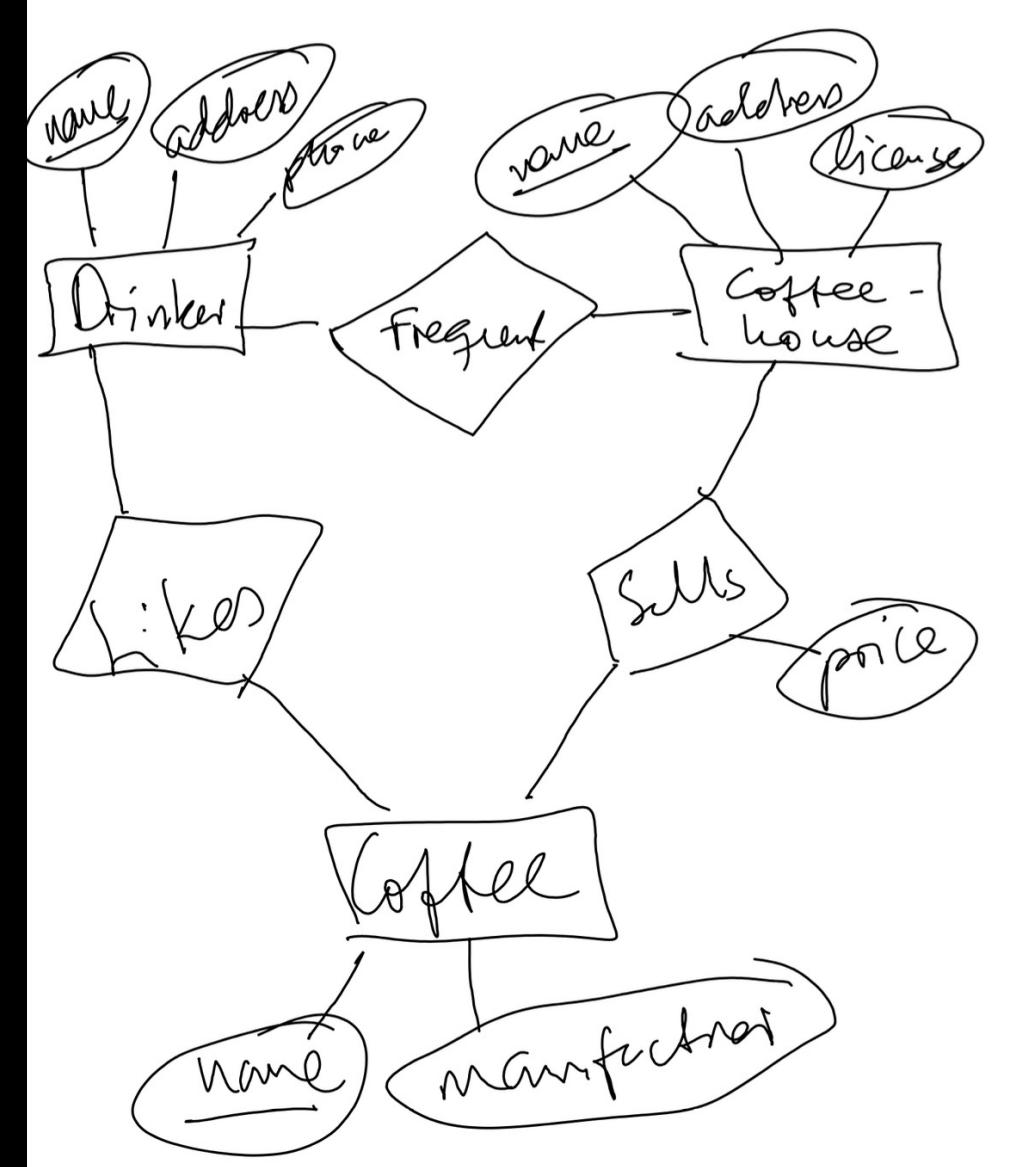
- Relationships may also have attributes



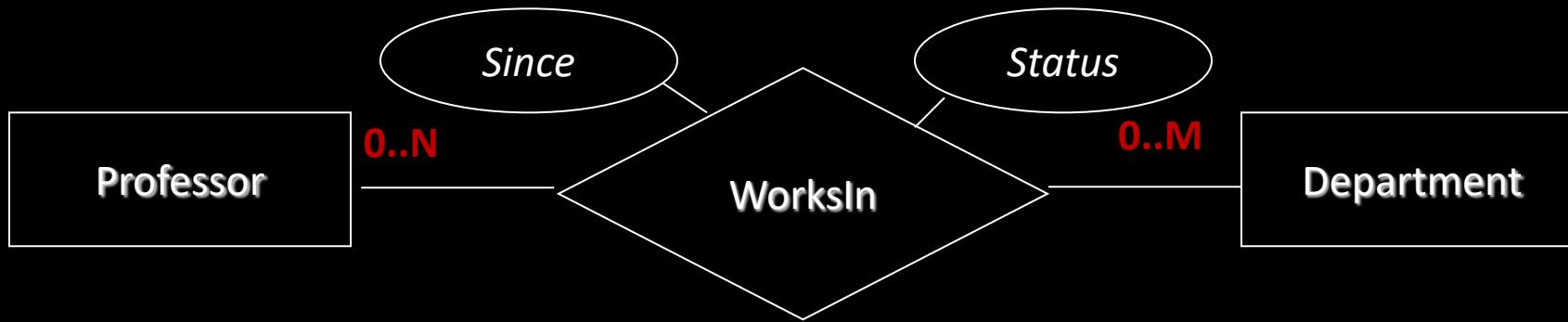
Example: Coffee Database

- What if ...

- price is an attribute of Coffee?
- price is an attribute of Coffeehouse?



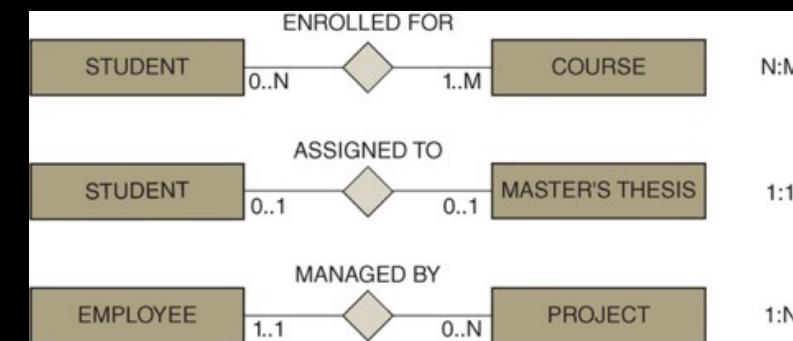
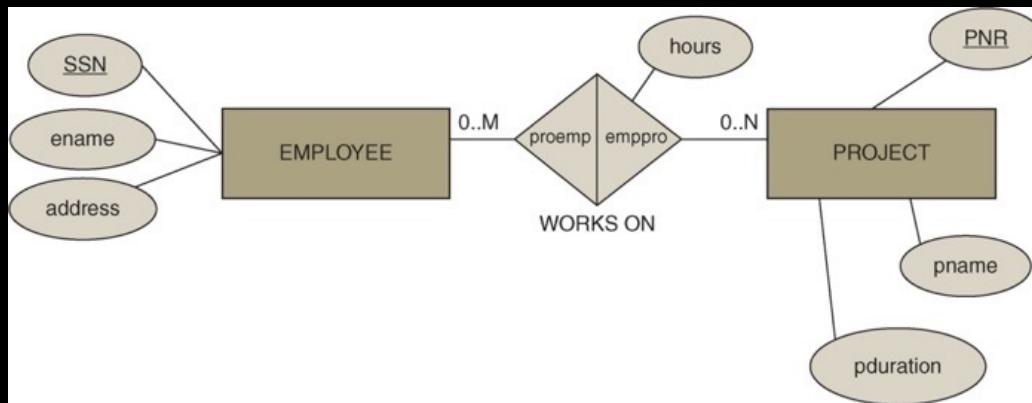
Basic Relationship Table in SQL DDL



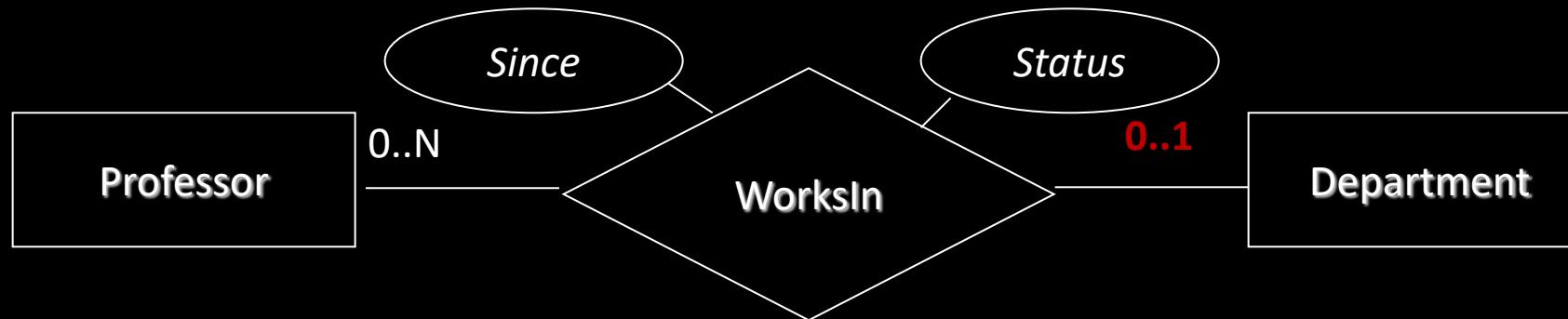
```
CREATE TABLE WorksIn (
    ProfID INTEGER,                                -- role (key of Professor)
    DeptID CHAR (4),                               -- role (key of Department)
    Since DATE NOT NULL,                           -- attribute
    Status CHAR (10) NOT NULL,                      -- attribute
    PRIMARY KEY (ProfID, DeptID),
    FOREIGN KEY (ProfID) REFERENCES Professor (ID),
    FOREIGN KEY (DeptID) REFERENCES Department (ID)
)
```

Cardinalities

- Relationships always have cardinalities
 - Minimum: 0 or 1
 - Maximum: 1 or N / M / L / * / ...
- • Read: Entity (ignore) Relationship Cardinality Entity
 - Student can enrol for 1 to M courses
 - Project is managed by exactly 1 employee
- • Do cardinalities impact the resulting table structure?

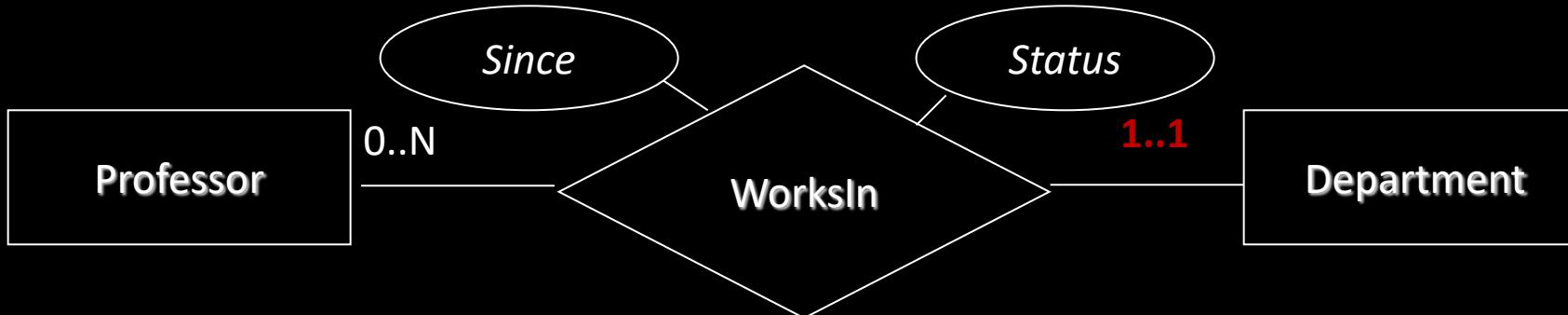


Maximum 1



```
CREATE TABLE WorksIn (
    ProfID INTEGER,          -- role (key of Professor)
    DeptID CHAR (4) NOT NULL, -- role (key of Department)
    Since DATE NOT NULL,     -- attribute
    Status CHAR (10) NOT NULL, -- attribute
    PRIMARY KEY (ProfID),   -- each professor only once
    FOREIGN KEY (ProfID) REFERENCES Professor (ID),
    FOREIGN KEY (DeptID) REFERENCES Department (ID)
)
```

Exactly 1

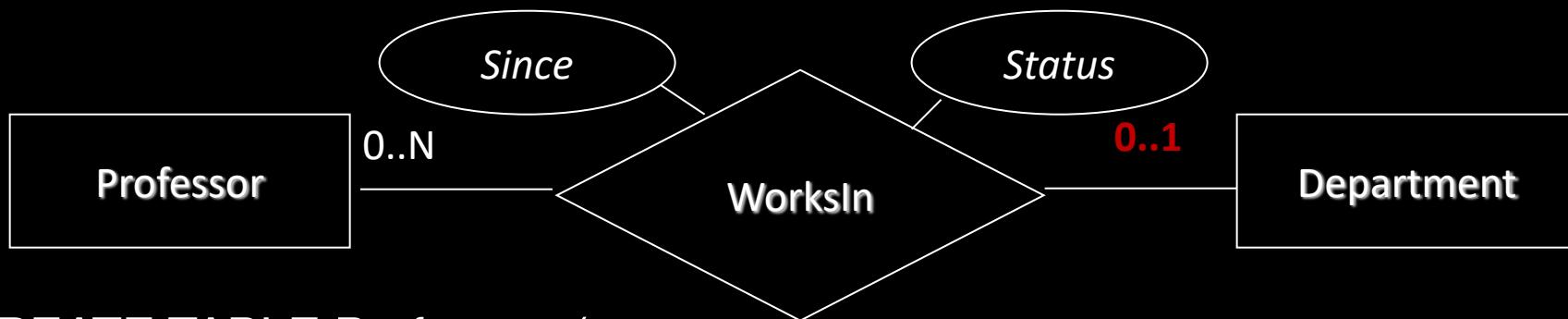


~~CREATE TABLE WorksIN~~

```
CREATE TABLE Professor (
    Id INTEGER PRIMARY KEY,
    ...
    deptId INTEGER NOT NULL,    -- foreign key
    Since DATE NOT NULL,        -- attribute
    Status CHAR (10) NOT NULL,   -- attribute
    FOREIGN KEY (deptId) REFERENCES Department(Id)
)
```

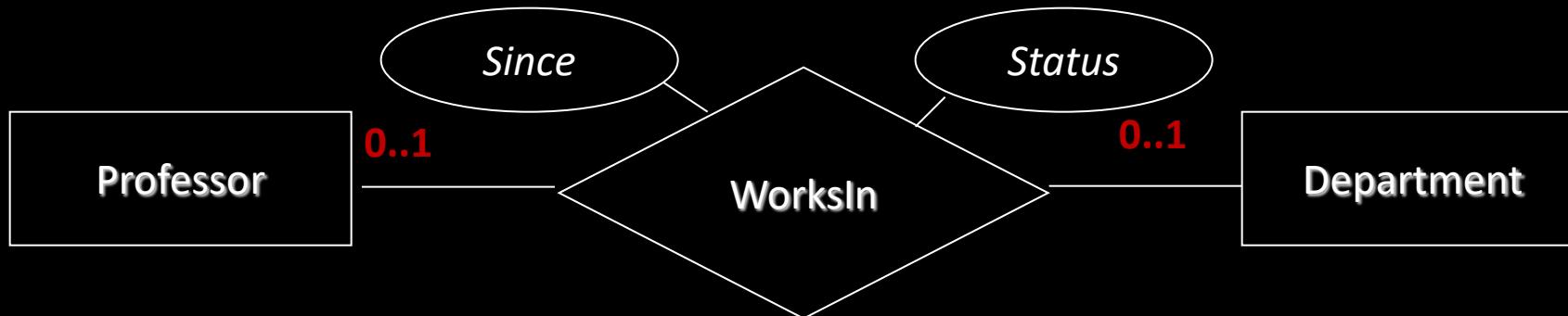
Maximum 1 - revisited

- Could we do the same with 0..1?
 - Yes, but three attributes must all be NULL or not NULL
 - We only do this if the relationship has NO attributes!!



```
CREATE TABLE Professor (
    Id INTEGER PRIMARY KEY,
    ...
    deptId INTEGER NULL, -- foreign key
    Since DATE NULL, -- attribute
    Status CHAR (10) NULL, -- attribute
    FOREIGN KEY (deptId) REFERENCES Department(Id)
)
```

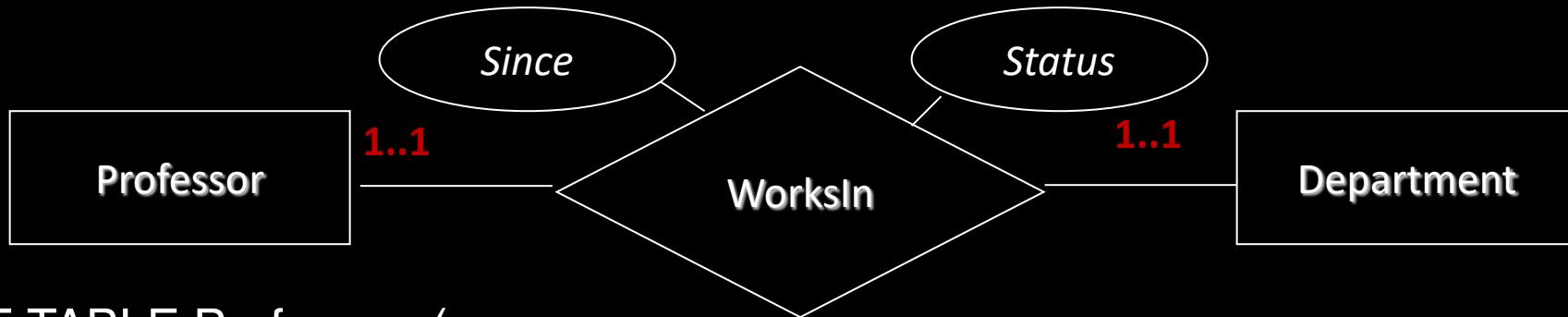
Maximum 1 - in both directions!



```
CREATE TABLE WorksIn (
    ProfID INTEGER,          -- role (key of Professor)
    DeptID CHAR (4) NOT NULL, -- role (key of Department)
    Since DATE NOT NULL,     -- attribute
    Status CHAR (10) NOT NULL, -- attribute
    PRIMARY KEY (ProfID),   -- each professor only once
    UNIQUE (DeptID),        -- each department only once
    FOREIGN KEY (ProfID) REFERENCES Professor (ID),
    FOREIGN KEY (DeptID) REFERENCES Department (ID)
)
```

Exactly 1 - in both directions!

- Can we use FKs on both sides?
 - Yes... but it is neither easy nor portable



```
CREATE TABLE Professor (
    Id INTEGER PRIMARY KEY,
    ...
    deptId INTEGER NOT NULL,
    Since DATE NOT NULL,
    Status CHAR (10) NOT NULL,
    FOREIGN KEY (deptId)
    REFERENCES Department(Id)
)
```

```
CREATE TABLE Department (
    Id INTEGER PRIMARY KEY,
    ...
    profId INTEGER NOT NULL,
    FOREIGN KEY (profId)
    REFERENCES Professor(Id)
)
```

Exactly 1 – in both directions!

- **Can we use FKs on both sides?**
 - Yes... but it is neither easy nor portable
- **Think about inserting the first prof and dept**
 - Which comes first, the chicken or the egg?
- **Alternative 1: Deferred FK or trigger**
 - Runs at the end of a transaction – many systems support neither!

```
CREATE TABLE Professor (
    Id INTEGER PRIMARY KEY,
    ...
    deptId INTEGER NOT NULL,
    Since DATE NOT NULL,
    Status CHAR (10) NOT NULL,
    FOREIGN KEY (deptId)
    REFERENCES Department(Id)
)
```

```
CREATE TABLE Department (
    Id INTEGER PRIMARY KEY,
    ...
    profId INTEGER NOT NULL,
    FOREIGN KEY (profId)
    REFERENCES Professor(Id)
)
```

Exactly 1 - in both directions!

- **Alternative 2:
Merge tables**

- May work well in some cases
- Depends on the entities

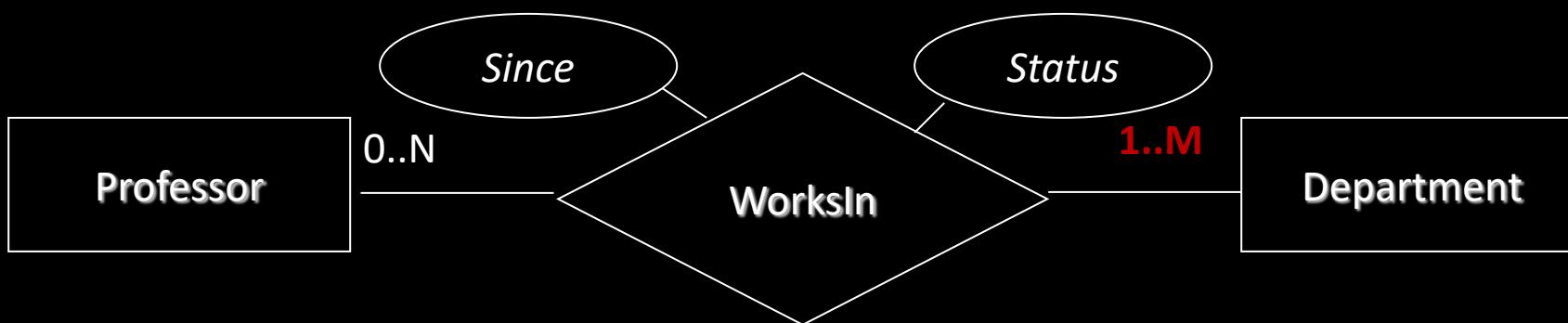
- **Alternative 3:
Pick one FK direction**

- Write down the other requirement
- Do the best we can in software with the other direction

```
CREATE TABLE Professor (
    Id INTEGER PRIMARY KEY,
    ...
    deptId INTEGER NOT NULL,
    deptName VARCHAR NOT NULL,
    ...
    Since DATE NOT NULL,
    Status CHAR (10) NOT NULL,
)
```

Minimum 1 - Maximum \mathcal{N}

- What about 1..N or 1..M cardinalities?
 - Or when requirements demand particular fixed numbers?
- No support in SQL DDL
 - Could use trigger code on Professor/WorksIn
- Normally:
 - Write down the requirement
 - Do the best we can in software



Exercise

- **Draw this schema as ER diagram ... but:**
 - Use IDs
 - Assume each coffeehouse sells exactly one coffee
- **Write SQL DDL to create the tables**
 - How many tables?

Coffees(name, manf)

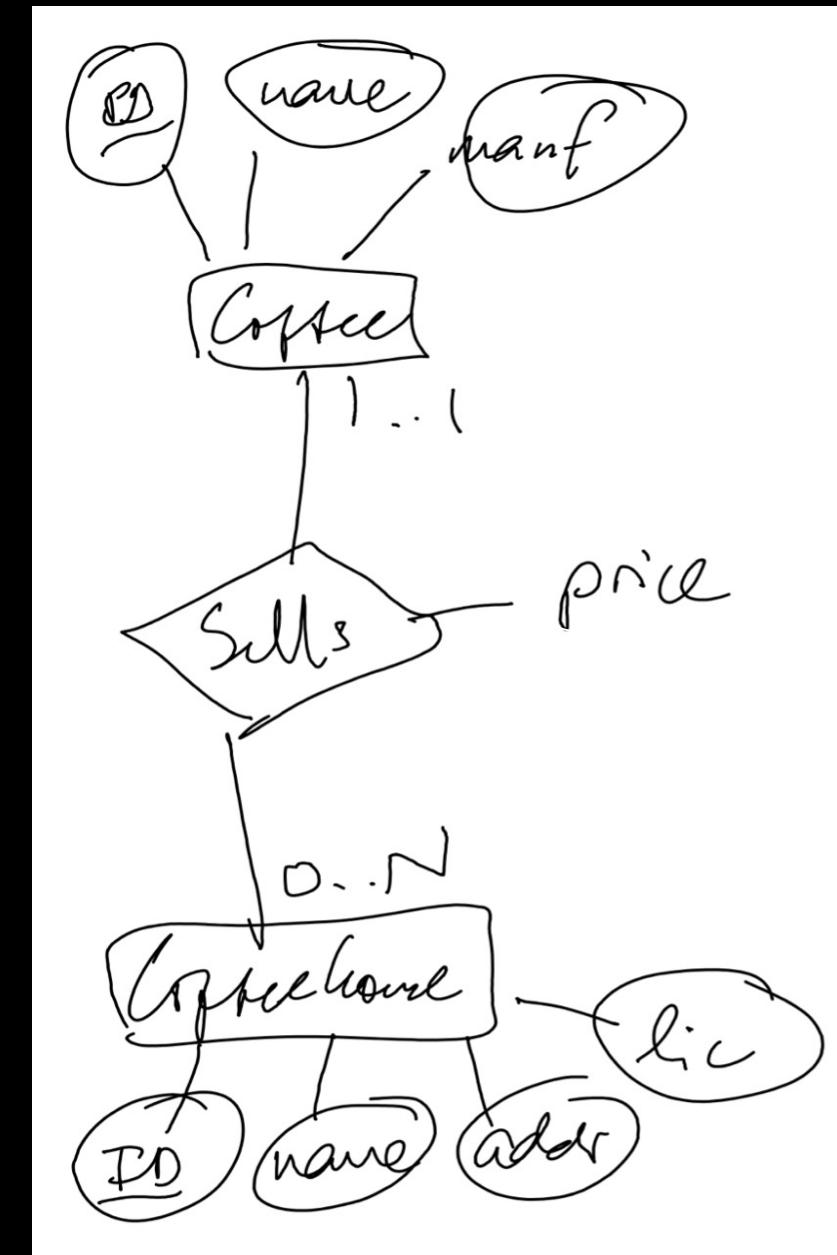
Coffeehouses(name, addr, license)

Sells(coffeehouse, coffee, price)

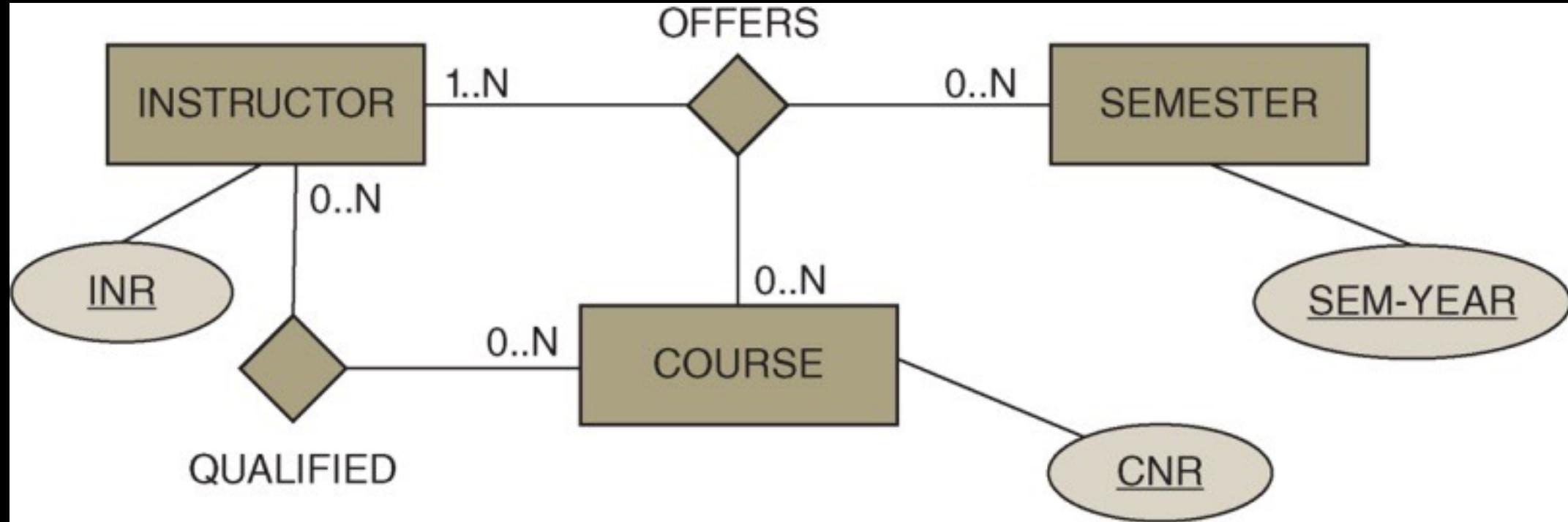
Answer

```
CREATE TABLE Coffee (
    ID INTEGER PRIMARY KEY,
    name VARCHAR NOT NULL,
    manf VARCHAR NOT NULL
);
```

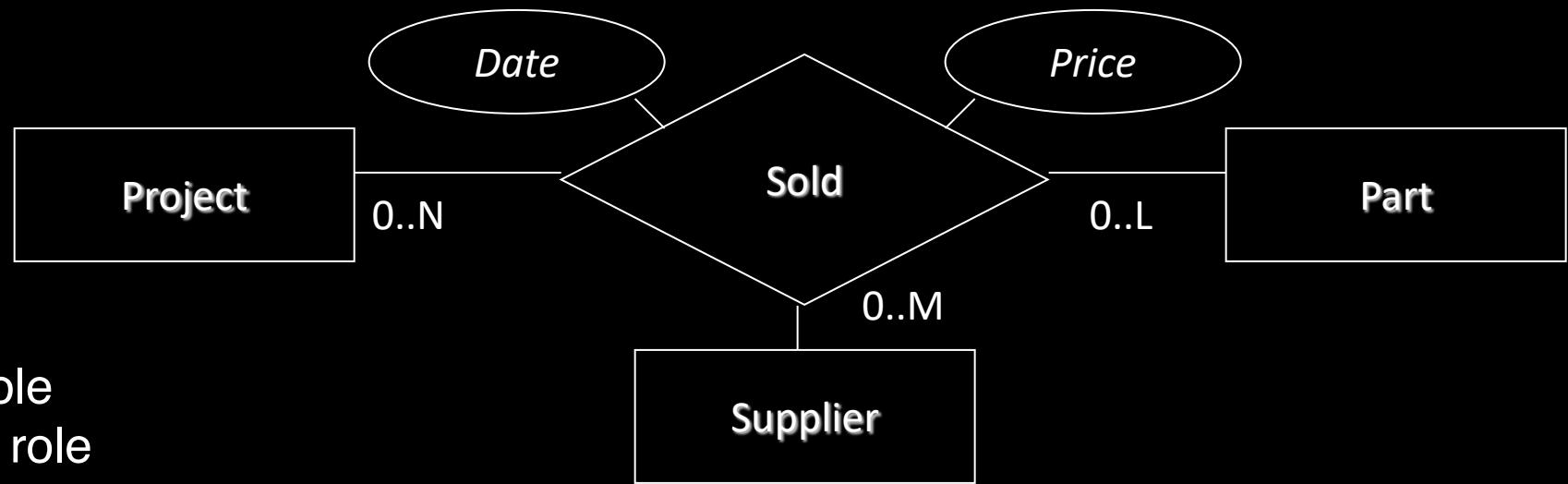
```
CREATE TABLE Coffeehouse (
    ID INTEGER PRIMARY KEY,
    name VARCHAR NOT NULL,
    addr VARCHAR NOT NULL,
    lic VARCHAR NOT NULL,
    coffeeID INTEGER NOT NULL
        REFERENCES Coffee(ID),
    price INTEGER NOT NULL
);
```



Tertiary Relationships (and Beyond)

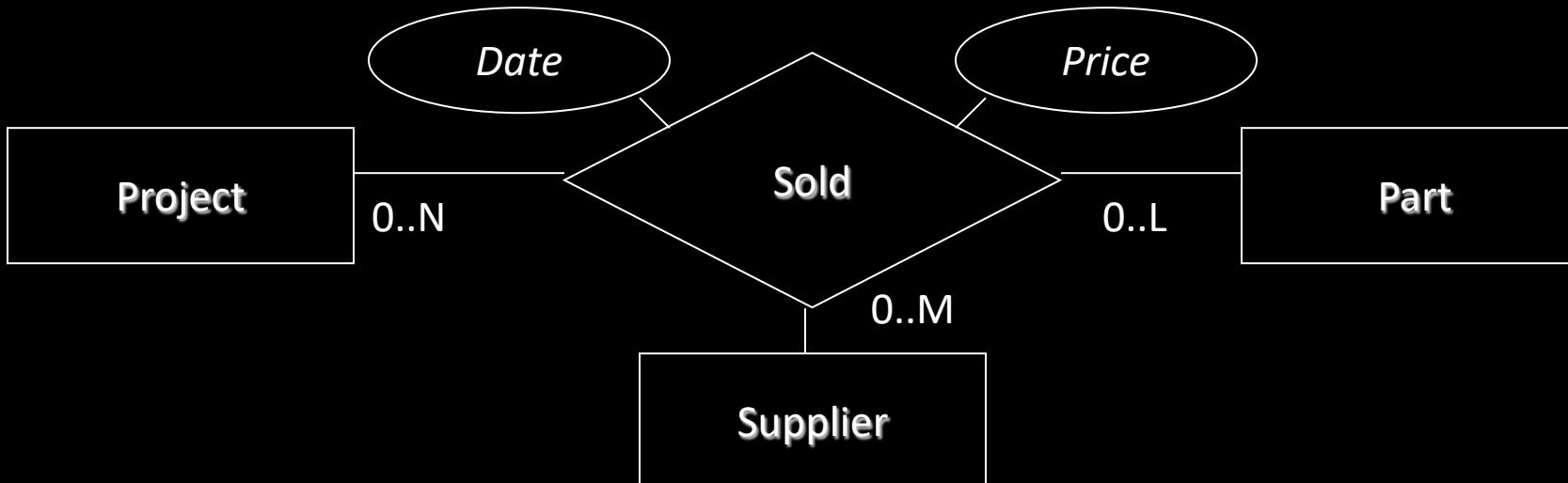


Example Tertiary Relationship Type



```
CREATE TABLE Sold (
    ProjID INTEGER,          -- role
    SupplierID INTEGER,      -- role
    PartNumber INTEGER,       -- role
    Date DATE NOT NULL,     -- attribute
    Price INTEGER NOT NULL   -- attribute
    PRIMARY KEY (ProjID, SupplierID, PartNumber),
    FOREIGN KEY (ProjID) REFERENCES Project (ID),
    FOREIGN KEY (SupplierID) REFERENCES Supplier (ID),
    FOREIGN KEY (PartNumber) REFERENCES Part
    (Number)
)
```

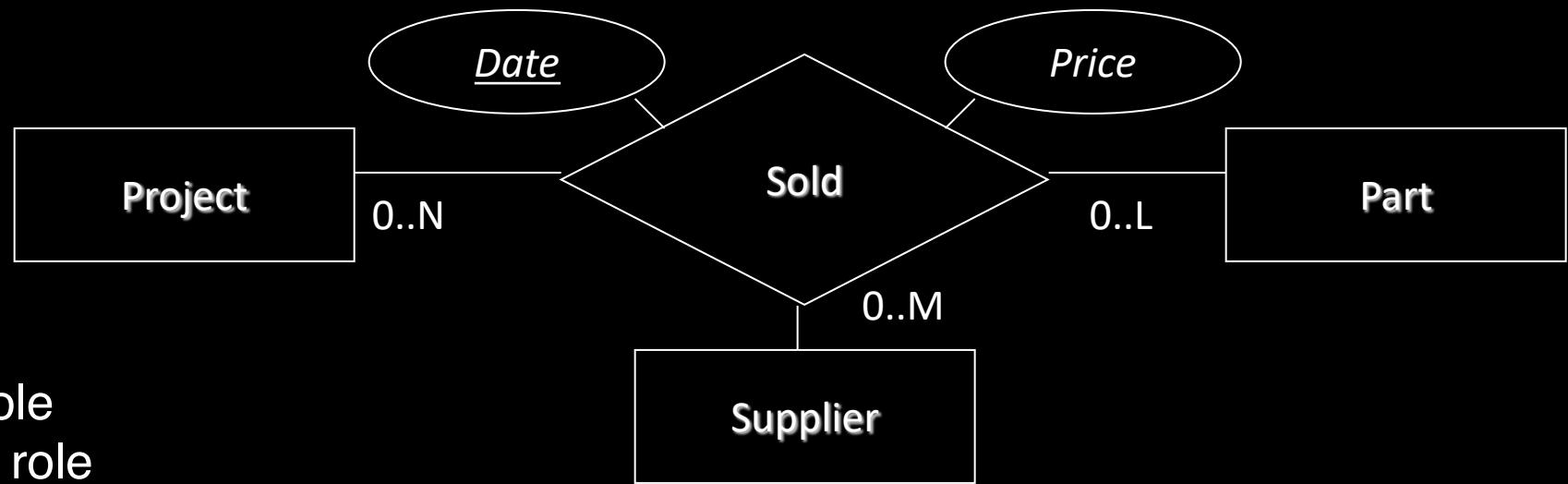
Extension: Partial Relationship Keys



PRIMARY KEY (ProjID, SupplierID, PartNumber)

- What does the key of the relationship table mean?
- What if the part should be sold many times?
- The book: No discussion!
- Our notation: Partial relationship key
 - Attribute is underlined

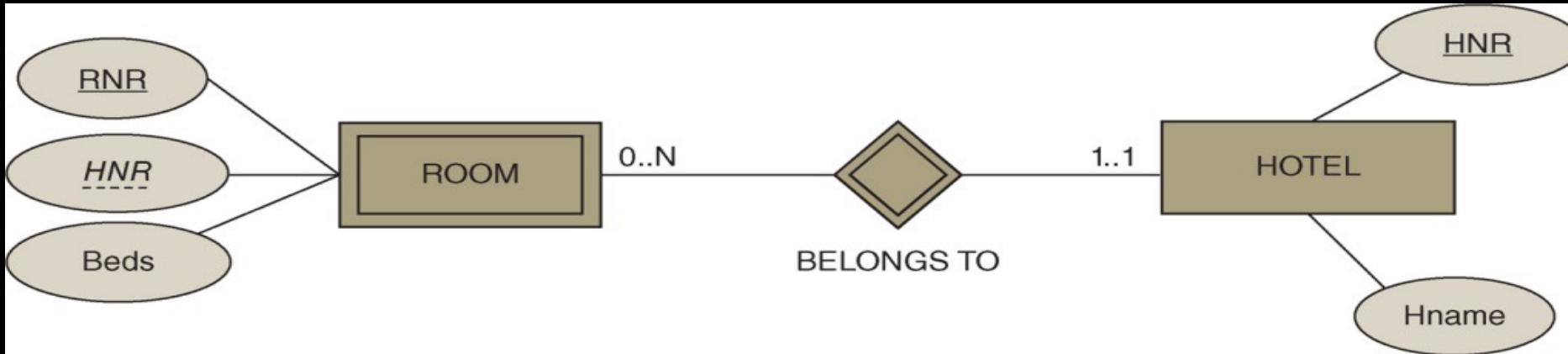
Partial Relationship Keys in SQL DDL



```
CREATE TABLE Sold (
    ProjID INTEGER,          -- role
    SupplierID INTEGER,      -- role
    PartNumber INTEGER,       -- role
    Date DATE,               -- attribute
    Price INTEGER NOT NULL,  -- attribute
    PRIMARY KEY (ProjID, SupplierID, PartNumber, Date),
    FOREIGN KEY (ProjID) REFERENCES Project (ID),
    FOREIGN KEY (SupplierID) REFERENCES Supplier (ID),
    FOREIGN KEY (PartNumber) REFERENCES Part
    (Number)
)
```

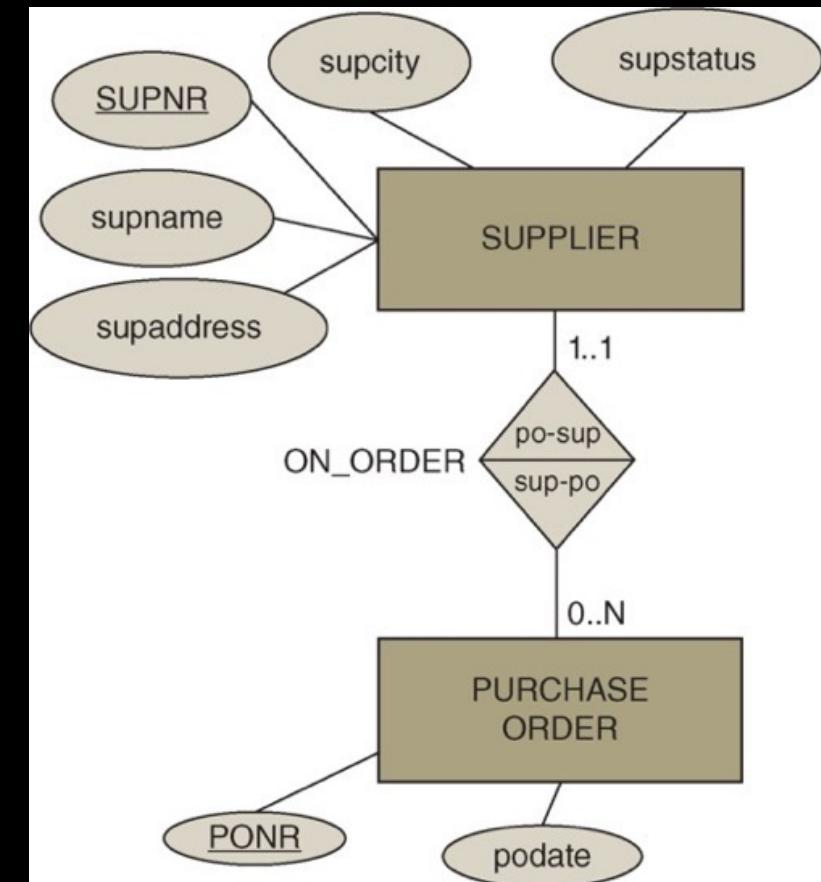
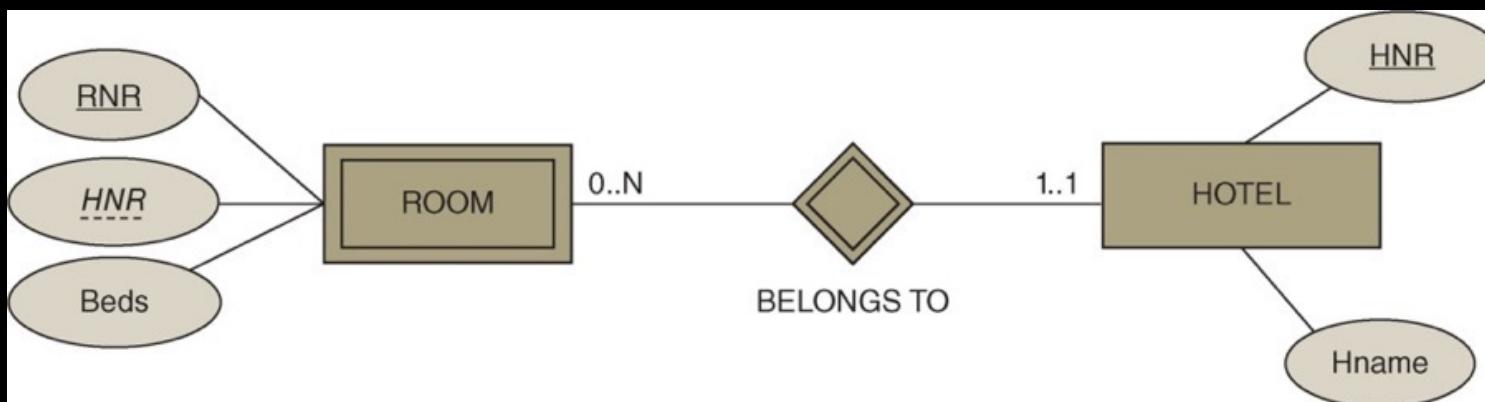
Weak Entity Types

- Weak entities belong to another entity
 - They do not have a proper key, but include “parent” key
 - They have a 1 .. 1 participation in the relationship
 - If “parent” is deleted, so is the “child”
- Representation
 - Double outlines (entity, relationship)
 - Parent key is underlined with dashes



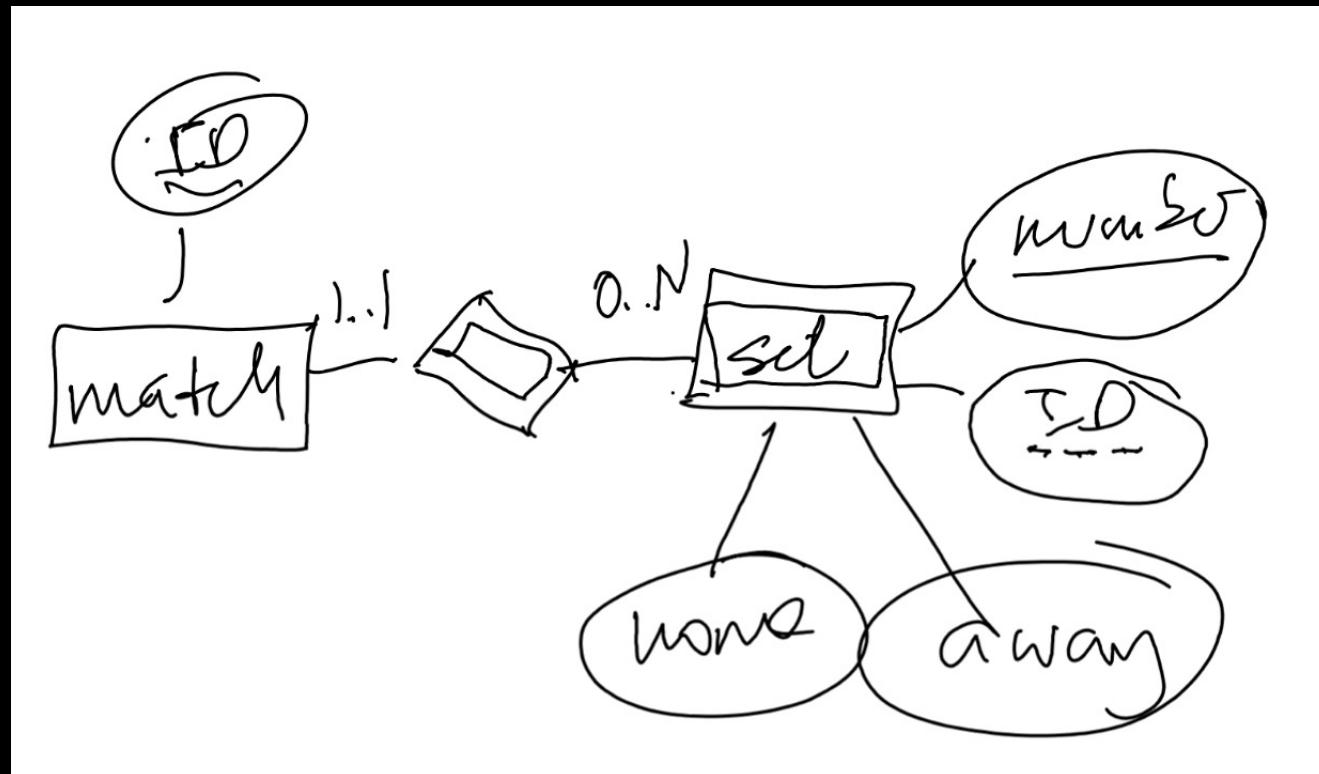
Weak Entities vs 1..1 Relationship Types

- Weak entities have a 1..1 relationship type
 - Some 1..1 relationship types represent weak entities
 - Most 1..1 relationship types do not represent weak entities
- Main difference is presence of a natural key
 - Weak entity:
Only unique within the parent entity!



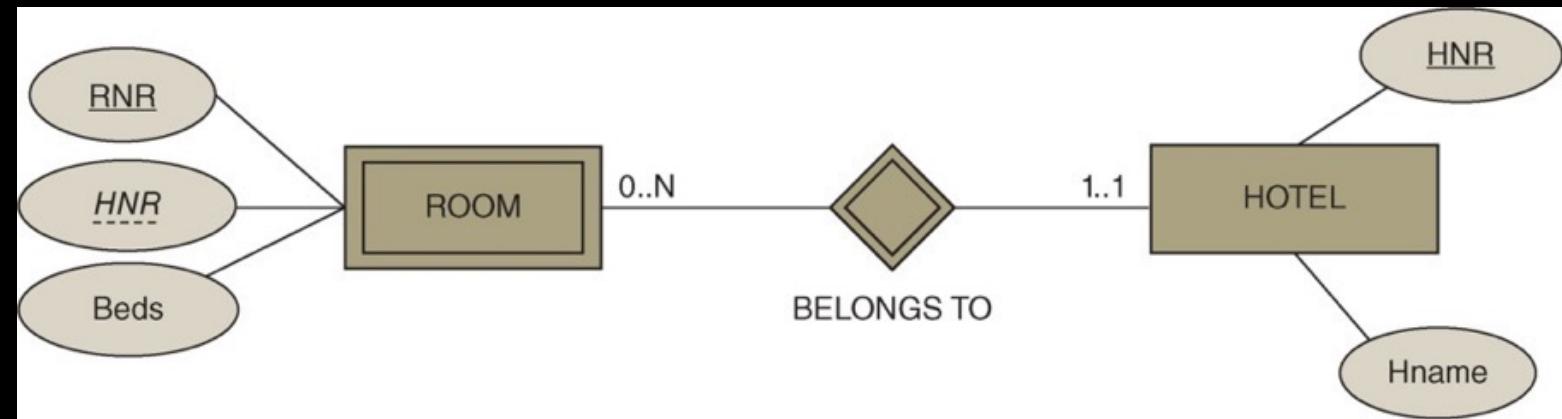
Practice: Weak Entity

- Each volleyball match consists of sets, each with set number, home score and away score



Weak Entities in SQL DDL

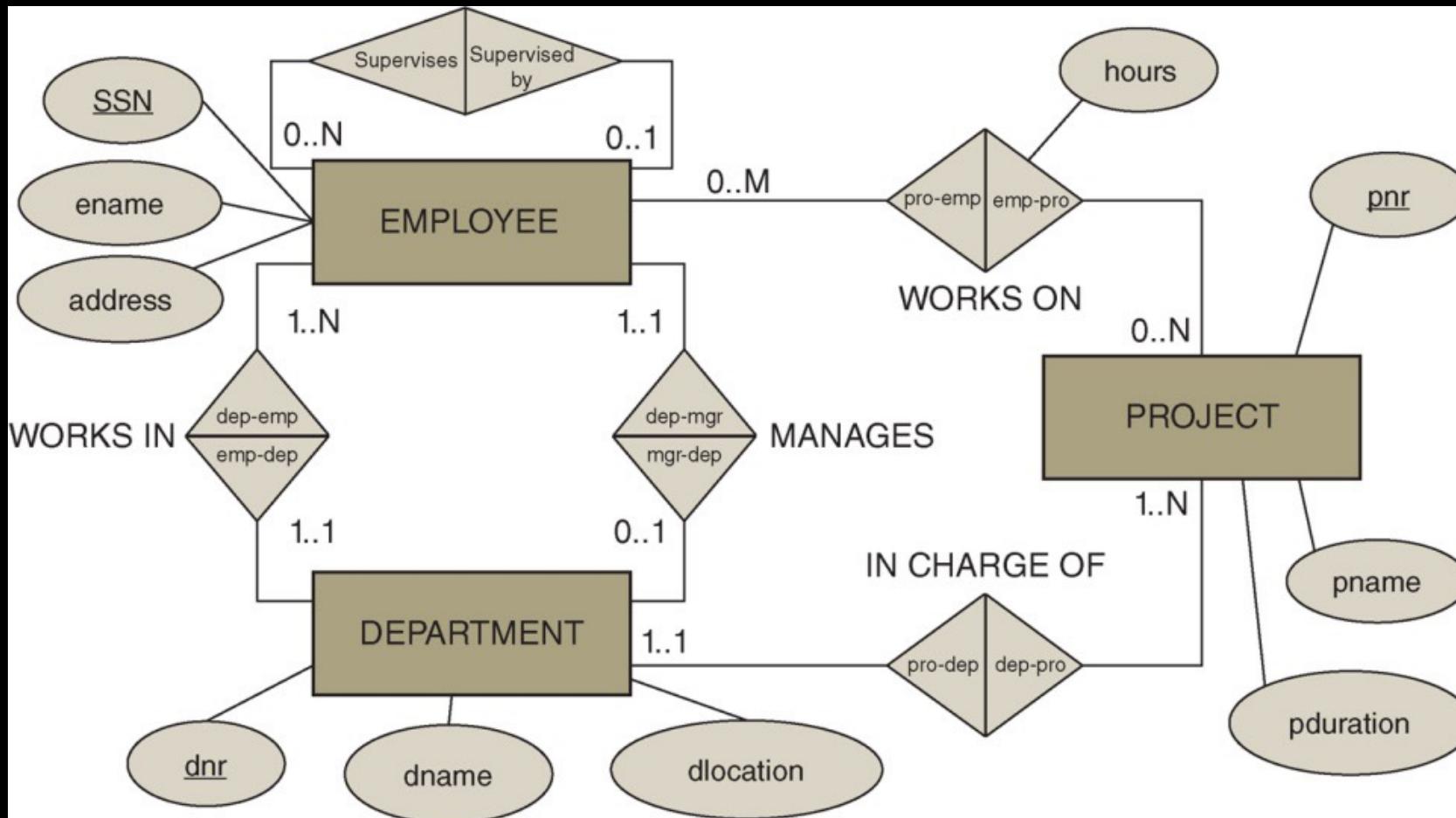
- Create a new table referencing the entity table
 - Primary key is parent key + partial key
 - Very similar to multi-valued attributes



```
CREATE TABLE Room (
    RNR INTEGER, -- Should not be SERIAL!
    HNR INTEGER -- Must be a FOREIGN KEY!
        REFERENCES Hotel(HNR),
    Beds INTEGER NOT NULL,
    PRIMARY KEY (HNR, RNR)
);
```

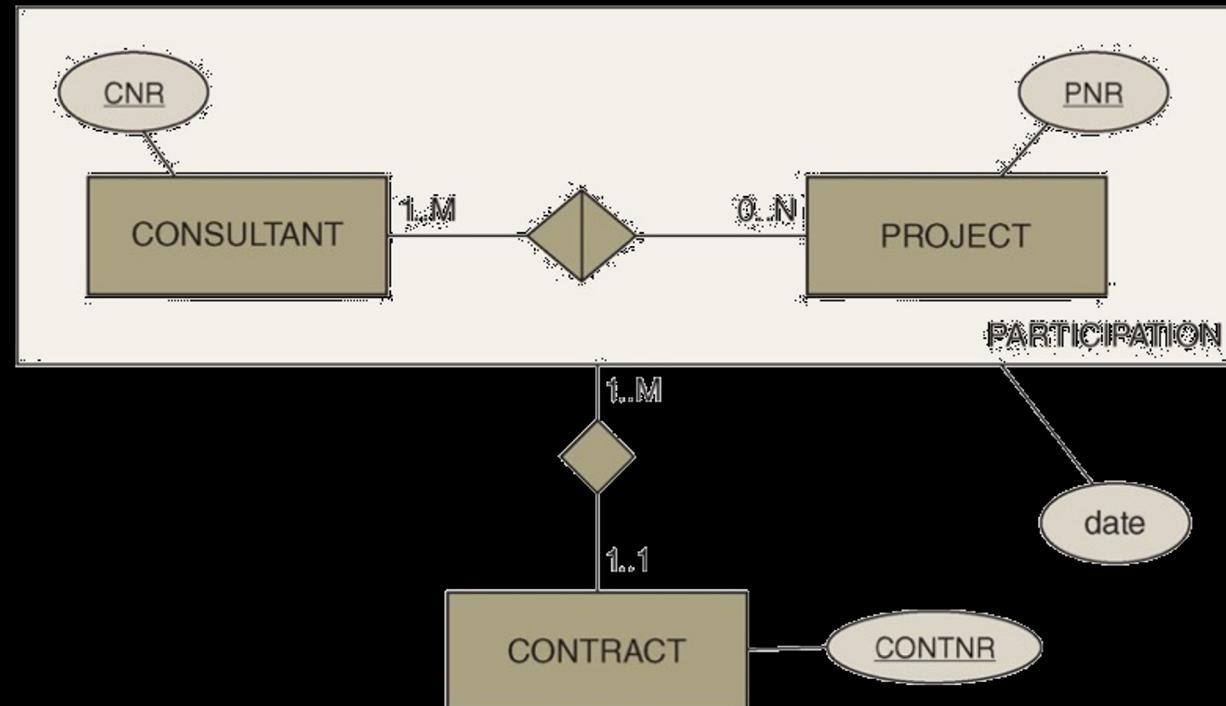
Exercise: Read ER Diagram

- How many tables will be created? What are the keys?

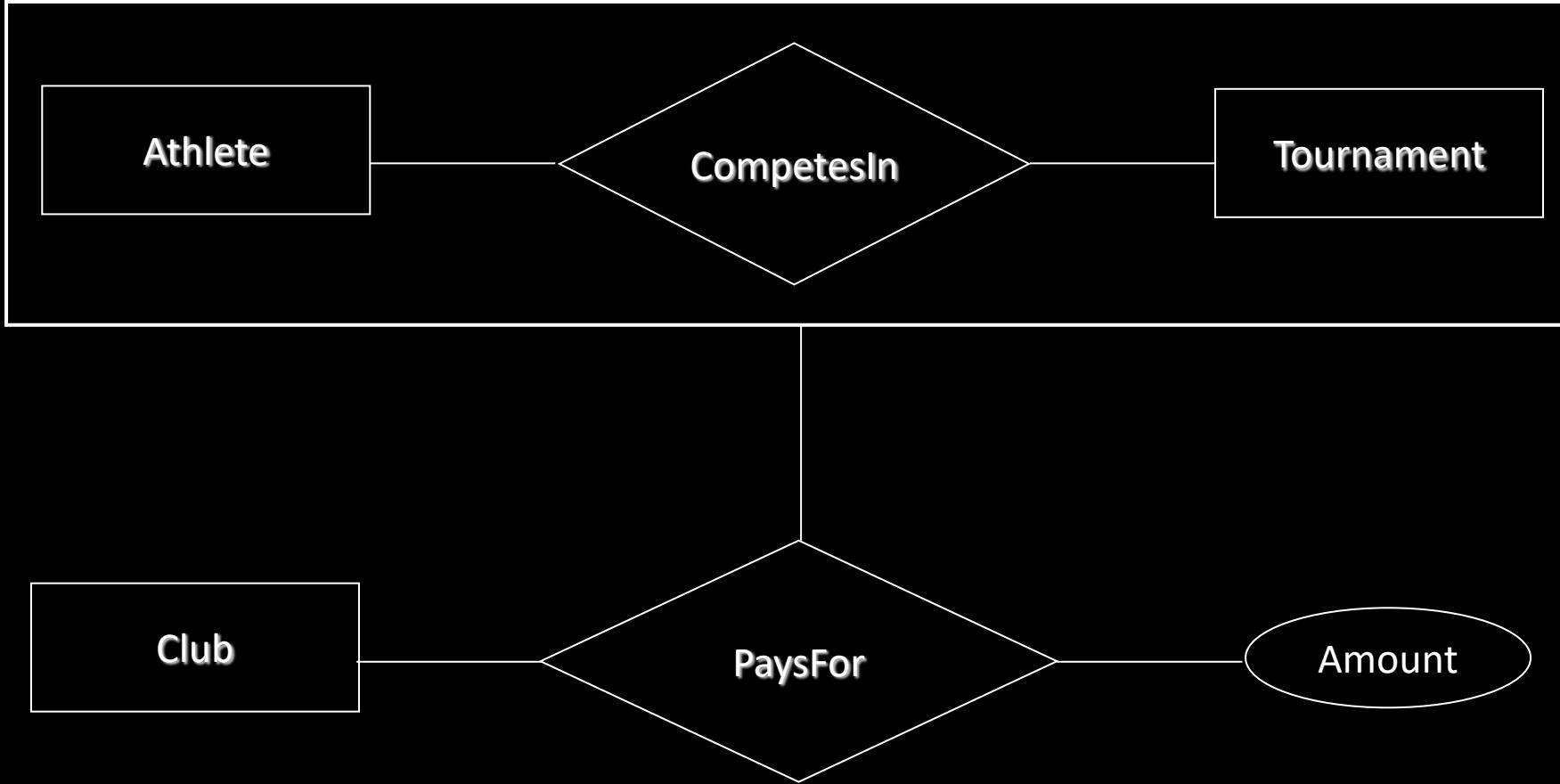


Aggregation

- Sometimes we need a relationship to a relationship
- Aggregation allows us to “convert” relationship types to entity types!
- Typical use: Monitoring, payments, contracts

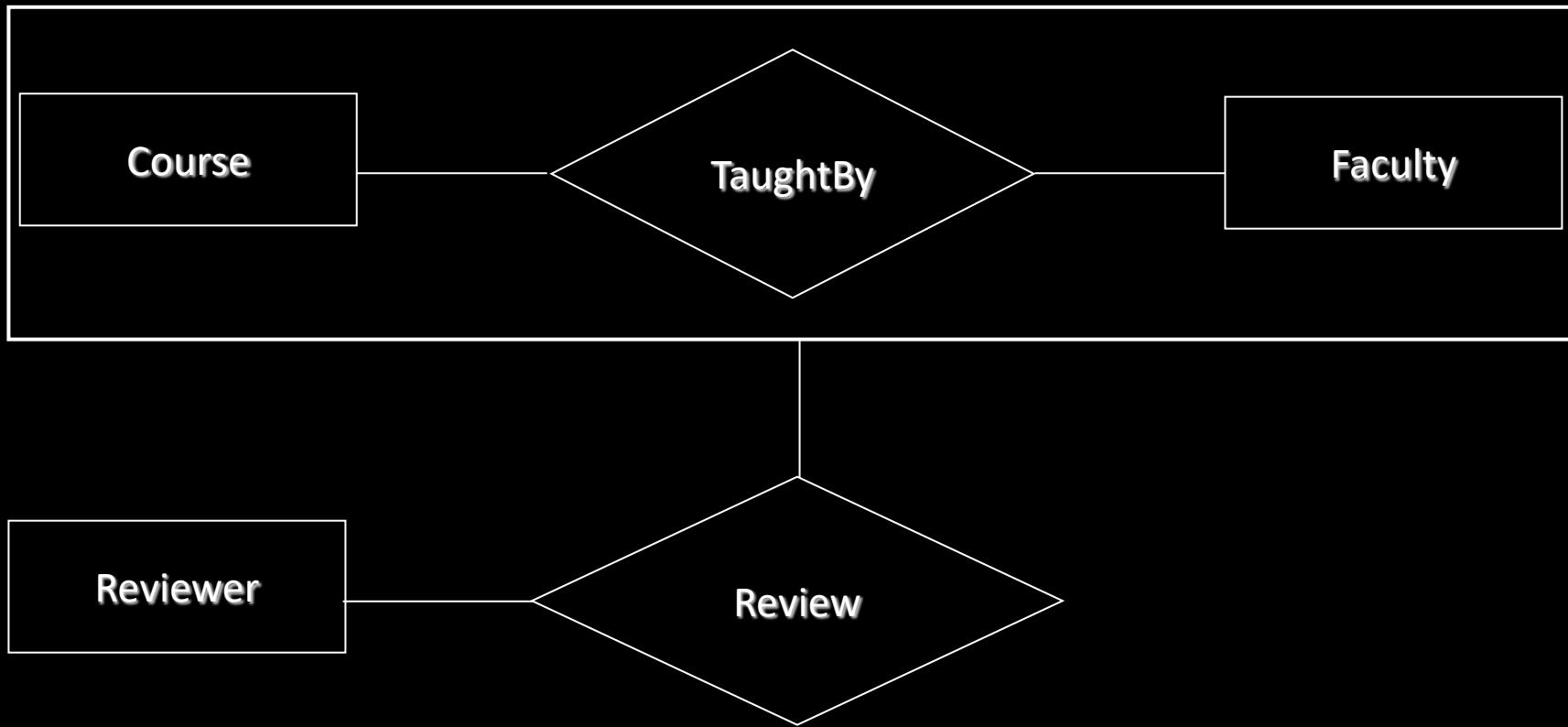


Aggregation: Relationship → Entity



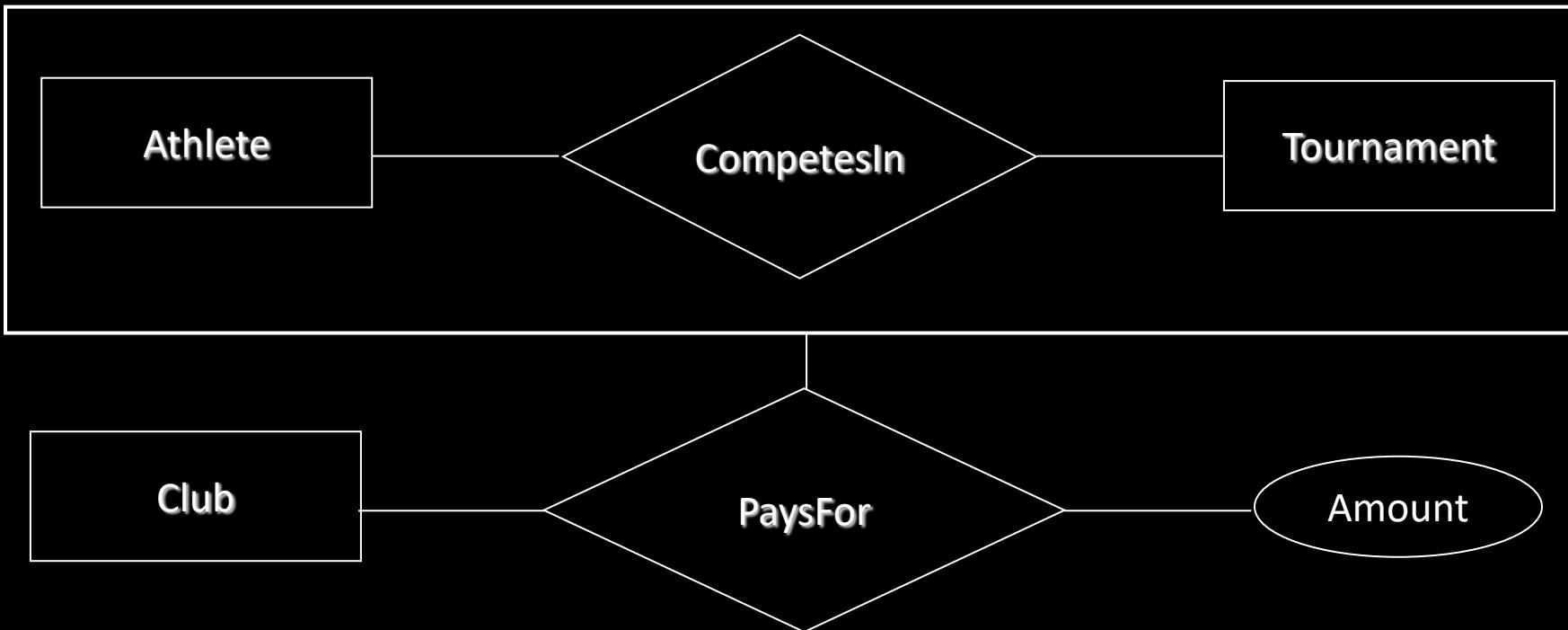
Aggregation Example

- Courses are taught by faculty members. External reviewers review each instance of the course.



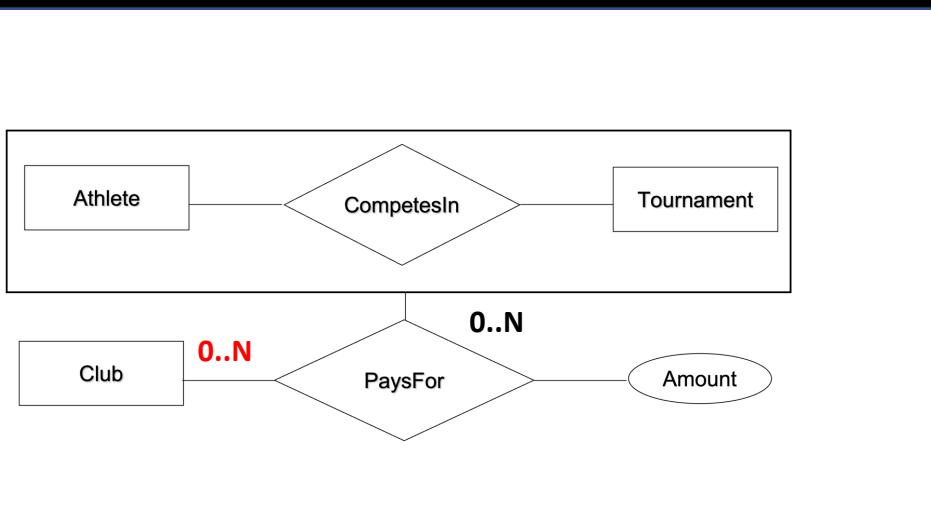
Translation to SQL DDL

- **Main observation: It is a relationship!**
 - Use the same method as for translating relationship
 - Treat the aggregation table as the entity!



Translation to SQL DDL: O..N

• Option 1: Use existing relationship key



```
create table CompetesIn (
    AID integer references Athletes,
    TID integer references Tournaments,
    primary key (AID, TID)
);

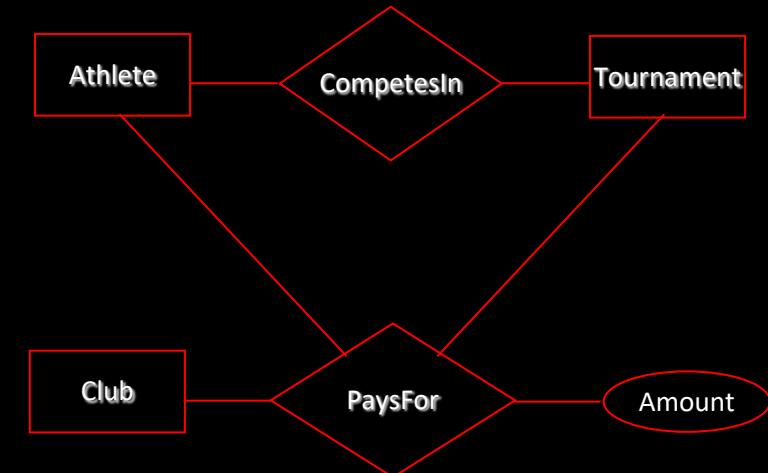
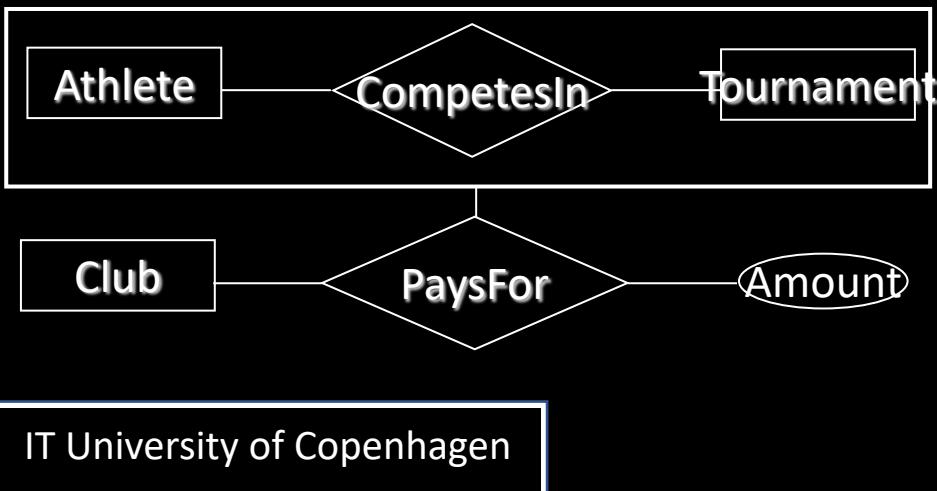
create table PaysFor (
    AID integer,
    TID integer,
    CID integer references Clubs,
    amount integer NOT NULL,
    foreign key (AID, TID) references CompetesIn
    (AID, TID),
    primary key (AID, TID, CID)
);
```

Option 1: Impact on Tables?

- A common error is to use FKs to the entity tables!

- This is actually equivalent to the ER diagram on the right
- See SQL code!

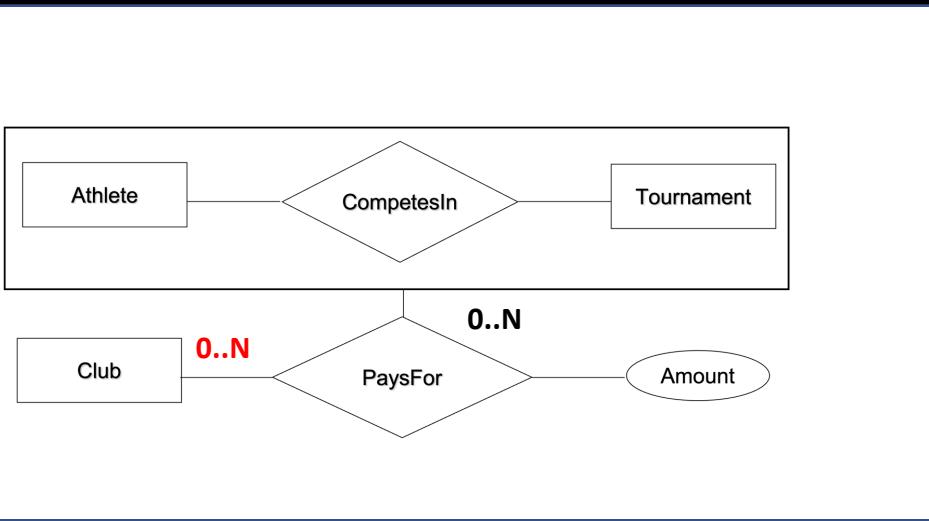
```
create table PaysFor (
    AID integer REFERENCES Athletes (AID),
    TID integer REFERENCES Tournaments (TID),
    CID integer references Clubs,
    amount integer NOT NULL,
    PRIMARY KEY (AID, TID, CID)
    FOREIGN KEY (AID, TID) REFERENCES CompetesIn
    (AID, TID),
);
```



Translation to SQL DDL: o..N

● Option 2: Create a new relationship key

- A common error is to then forget about the existing key!



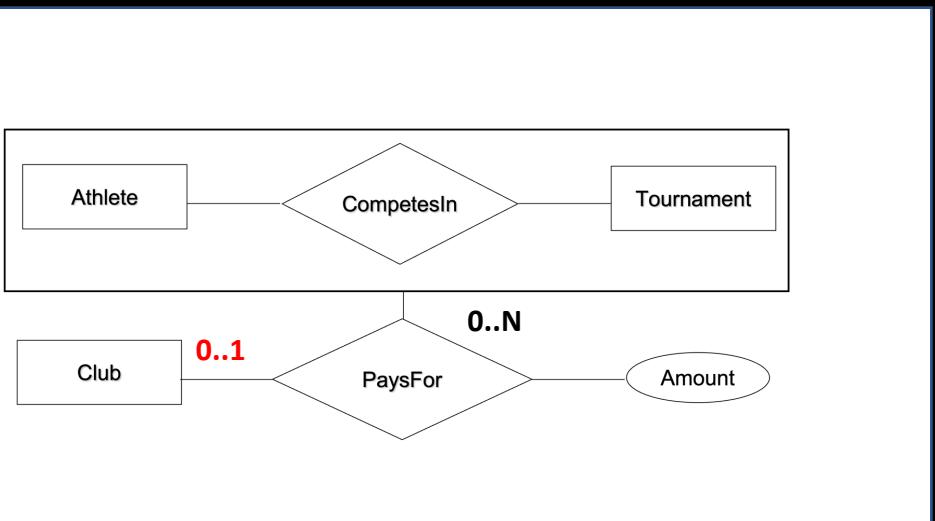
```
create table CompetesIn (
    CIID SERIAL primary key,
    AID integer NOT NULL references Athletes,
    TID integer NOT NULL references Tournaments,
    UNIQUE (AID, TID)
);
```

```
create table PaysFor (
    CIID integer references CompetesIn,
    CID integer references Clubs,
    amount integer NOT NULL,
    primary key (CIID, CID)
);
```

Translation to SQL DDL: 0..1

● Change the primary key

- This might happen when payment comes later



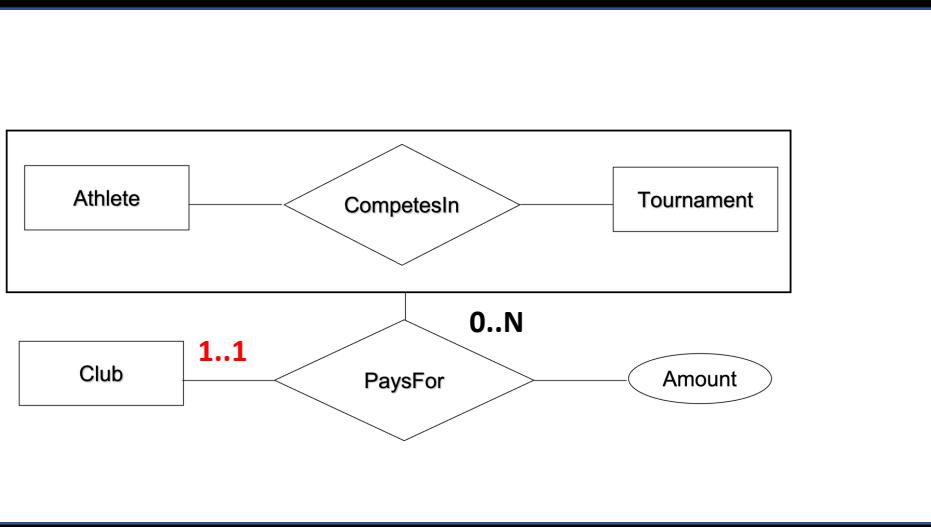
```
create table CompetesIn (
    AID integer references Athletes,
    TID integer references Tournaments,
    primary key (AID, TID)
);

create table PaysFor (
    AID integer,
    TID integer,
    CID integer NOT NULL references Clubs,
    amount integer NOT NULL,
    foreign key (AID, TID) references CompetesIn (AID,
    TID),
    primary key (AID, TID)
);
```

Translation to SQL DDL: 1..1

● Change the relationship table

- This is the only case covered in the book!
- Here: No PaysFor table!

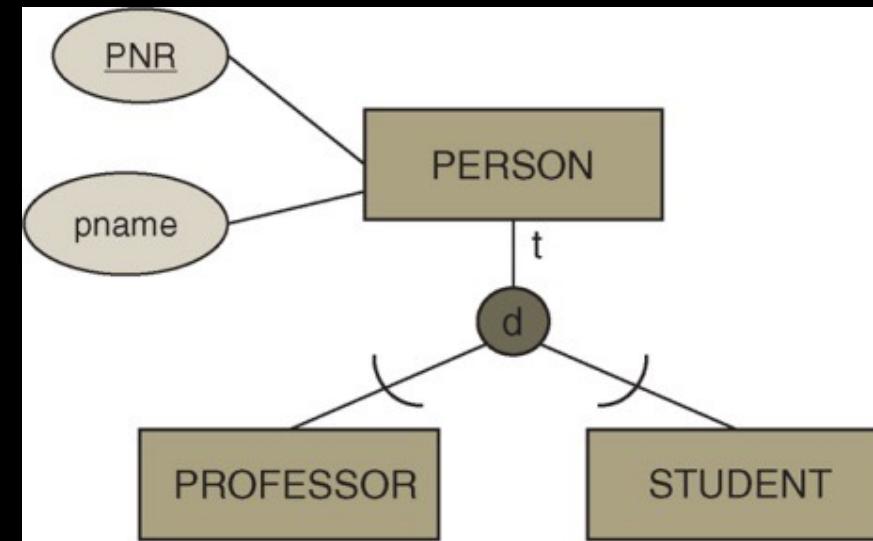
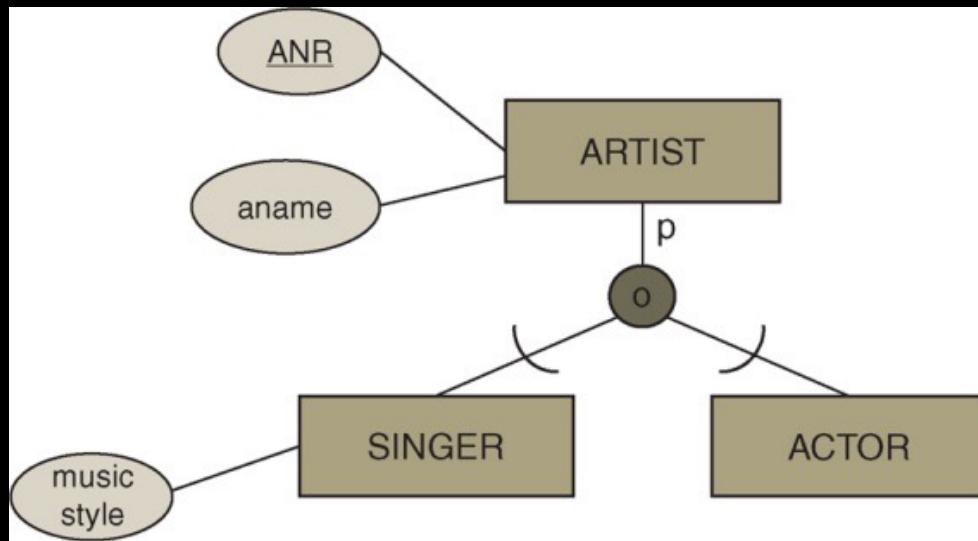


```
create table CompetesIn (
    AID integer references Athletes,
    TID integer references Tournaments,
    CID integer references Clubs NOT NULL,
    amount INTEGER NOT NULL,
    primary key (AID, TID)
);
```

Generalization/Specialization

- Like classes in Java

- Partial vs Total = p/t on the line
- Overlapping vs Disjoint = o/d in the circle
- Please don't use colour in homeworks/exam
- Arcs matter → need to draw them (in the exam)!



Specialization in SQL DDL

- **One table for super-type, one per sub-type**
 - The PK of supertype is also PK for all subtypes
 - Each subtype has a FK to the supertype
- **Option 1 in the PDBM book – preferred option by far!**
 - Redundancy is eliminated:
Name and DOB are stored only once
 - Adjusts well to hierarchies/lattices

Person			Employee			Student		
SSN	Name	DOB	SSN	Department	Salary	SSN	GPA	<u>StartDate</u>
1234	Mary	1950	1234	Accounting	35000	1234	3.5	1997

Specialization in SQL DDL: Option 1

```
CREATE TABLE Person (
    SSN INTEGER PRIMARY KEY,
    Name VARCHAR NOT NULL,
    DOB DATE NOT NULL
);
```

```
CREATE TABLE Employee (
    SSN INTEGER
        PRIMARY KEY
        REFERENCES Person(SSN),
    Department VARCHAR NOT NULL,
    Salary INTEGER NOT NULL
);
```

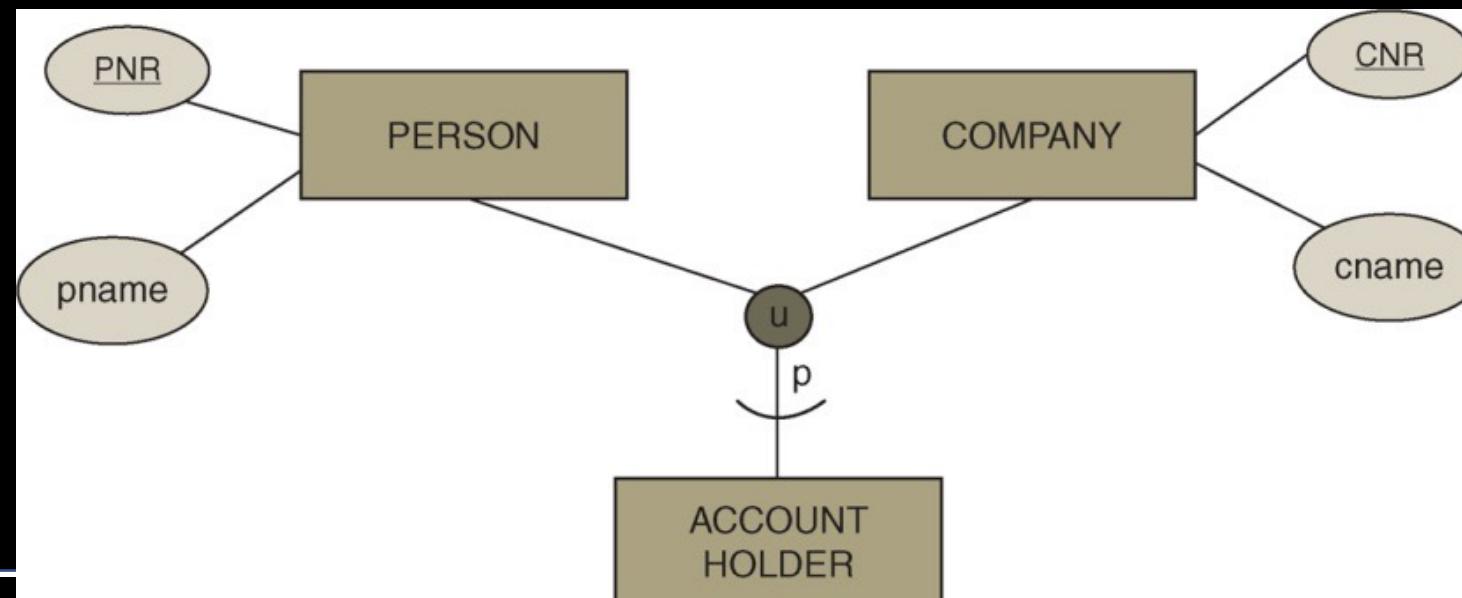
```
CREATE TABLE Student (
    SSN INTEGER
        PRIMARY KEY
        REFERENCES Person(SSN),
    GPA REAL NOT NULL,
    StartDate DATE NOT NULL
);
```

Person			Employee			Student		
SSN	Name	DOB	SSN	Department	Salary	SSN	GPA	<u>StartDate</u>
1234	Mary	1950	1234	Accounting	35000	1234	3.5	1997

Categorization

- **Grouping of otherwise unrelated entities**

- New entity is a union = u in the circle
 - Can be total or partial = p/t on the line
- Can be checked with triggers similarly to specialization
- Notice the direction of the arc to the union
 - Refers to flow of “inheritance” – or which comes first...
 - Arcs matter → need to draw them (in the exam)!



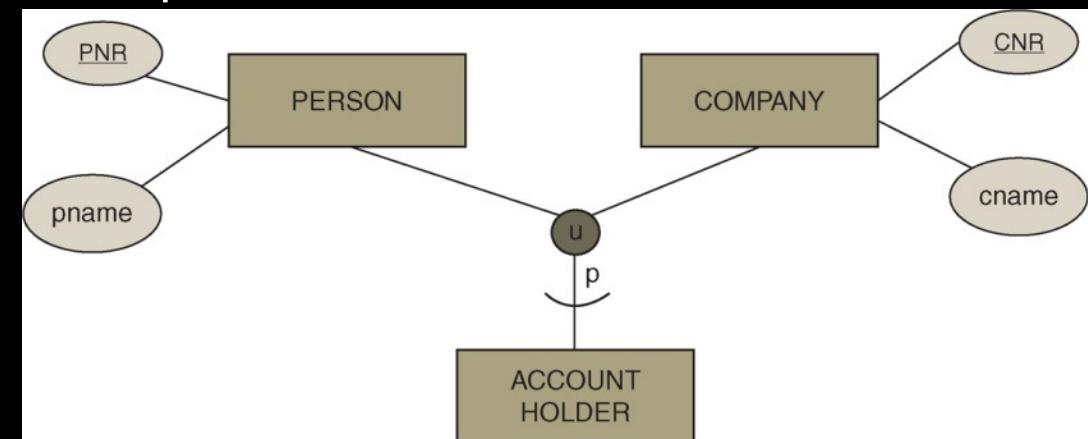
Categorization in SQL DDL

- New table for the categorical entity
 - New abstract PK with INTEGER / SERIAL
- Add the PK as attribute to the other entities
 - With FK to the categorical entity
- Why is this important?

Sometimes AccountHolder participates in relationships...

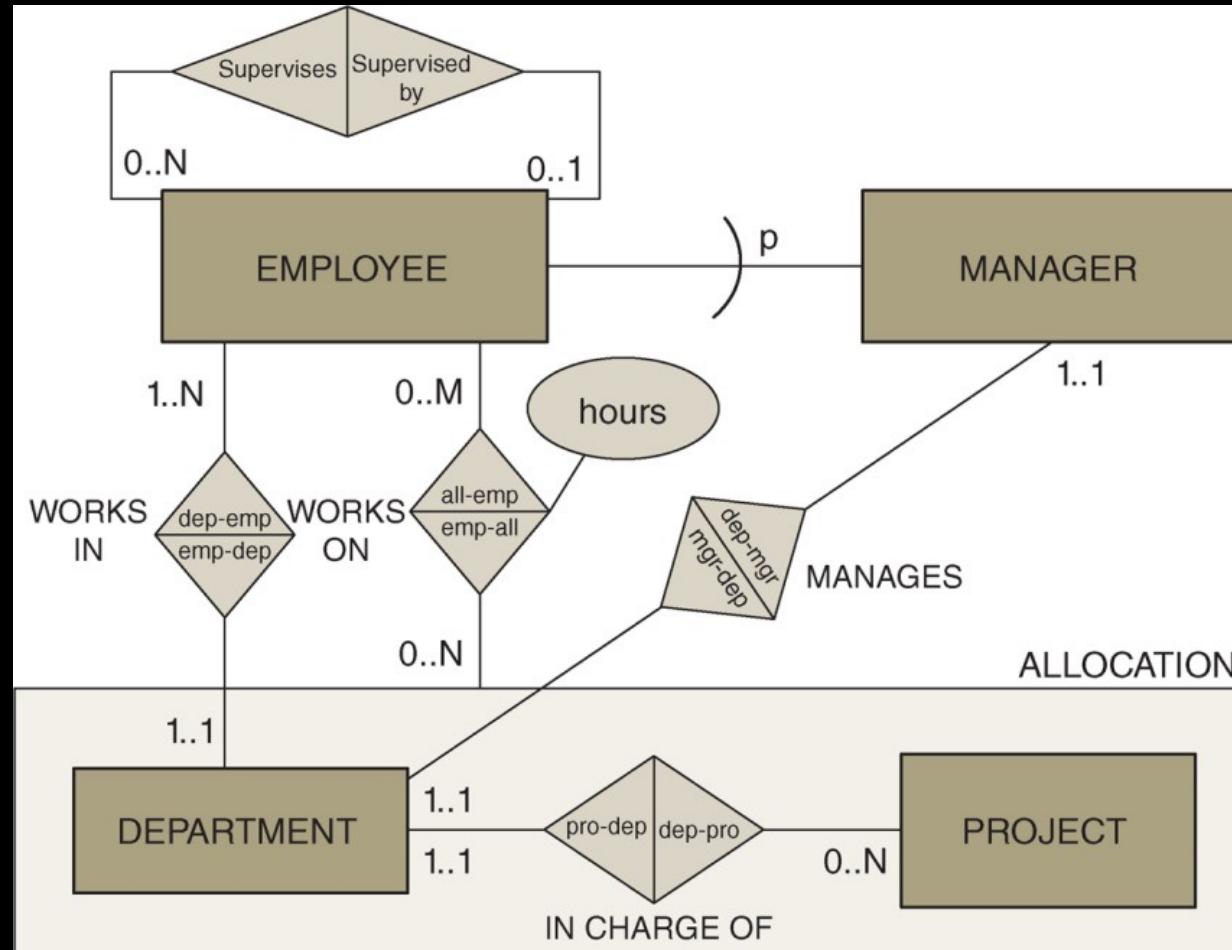
```
CREATE TABLE AccountHolder (
    AcctID SERIAL PRIMARY KEY
);
```

```
CREATE TABLE Person (
    PNR SERIAL PRIMARY KEY,
    pname VARCHAR NOT NULL,
    AcctID INTEGER [NOT NULL] REFERENCES AccountHolder(AcctID)
);
```



Practice: Read EER diagram

- How many tables will be created? What are the keys?



Getting Started?

- **Nouns = entities**
 - Descriptive elements = attributes
 - **Verbs = relationships**
 - Descriptive elements = attributes
 - Look for words implying participation constraints
 - No words → 0..N
-
- Professors have an SSN, a name, an age, a rank, and a research specialty.
 - Projects have a project number, a sponsor name (e.g., NSF), a starting date, ...
 - Each project is managed by one professor (1..1 on profs, 0..N on proj)
 - Professors may work on many projects (0..* on both sides)
 - Each project must be reviewed by some professors (1..N on profs, 0..N on proj)

Dealing with Very Large ER diagrams

- **Method 1: Very large paper!**
 - One diagram with all the details
- **Method 2: Outline + Details**
 - One diagram with main entities and their relationships
 - One diagram per entity with attributes and weak entities
- **Method 3: Components**
 - Break the model into components
 - Details inside components
 - Some edge entities are repeated (details in one place)

Limitations of ER Designs

- **ER diagrams do not capture all design details**
 - Example: Multiple candidate keys
 - Must note missing details somewhere!
- **Some aspects do not map well to SQL DDL**
 - Example: 1..M cardinalities
 - Triggers (week 4, BSc) can be used to handle some problems
 - Normalization (week 6) provides a mechanism for fixing some problems
 - Some must simply be noted and addressed in code or ignored!

Summary of Notation Extensions

- **Relationship roles are generally unnecessary**
 - No need to label roles
 - Except for unary relationship types
 - May put relationship name inside rhombus
 - Except for unary relationship types
- **We allow partial keys of relationships**
 - Underlined relationship attribute
 - Part of the PK of the resulting relationship relation
- **Aggregation entity may cover only the relationship**
 - Much easier to read!
- **Allow 0..* in place of 0..N/M/L**
 - ... or simply use 0..N everywhere!



Takeaways

- **ER design captures entities and relationships**
 - Weak entities, specialization, categorization and aggregation allow capturing more detailed model characteristics
 - This is hard – but also useful – so you must practice!
- **Conversion to SQL DDL**
 - Entities and relationships mapped to relations
 - Essentially an algorithmic process (with some options)
 - This is hard – but also useful – so you must practice!
- **Notation: MANY VARIANTS EXIST!**
 - We use the one from the book (as extended in lecture)
 - You must use this notation in the homework and exam!!!



What is next?

- **Next Lecture:**
 - Data Normalization

Introduction to Database Systems

I2DBS – Spring 2023

- Week 6:
- Normalization

Jorge-Arnulfo Quiané-Ruiz

Readings:

PDBM 6.2

Information

- **Homework 1:**
 - We provide feedback and solution this week
 - Also review document on LearnIT with common errors
- **Homework 2 due on March 24th at 23:59**
 - Use the exercise session to work on this!
 - **Remember:** 3 out of 4 mandatory; low bar for acceptance!
- **How will you submit ER diagrams in the exam?**
 - If you do it with software: Submit PNG/JPG/PDF file
 - If you do it by hand: Submit paper and proctors will scan
- **Exercise 7 (next week):**
 - Normalisation from this week (Lecture 6)
 - OS/Storage/Multicores from next week (Lecture 7)

Looking for an RA

- **Requirements:**
 - Good Java skills
 - Linux, macOS, windows experience
 - Highly motivated
- **Wish-List:**
 - Interested in open-source software
 - Looking for system building experience
 - Curious and creative developer
- **Position and Role**
 - Research Assistant for one year (half time)
 - Playing and having fun with Apache Wayang
 - Help us to develop and grow Apache Wayang
- **Interested?**
 - Send your CV and cover (motivation) letter to: joqu@itu.dk





Profile of the Week

Raymond A. Lorie

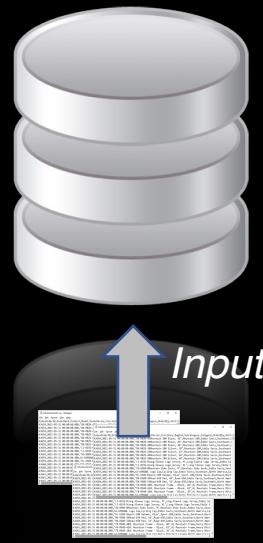
Pioneer of Normalization Theory

- 1936: Detroit, USA
- 1966: PhD in Electrical Engineering from MIT
- 1966: joined IBM Research in San Jose
- 1974: co-invented System R
- 1974: introduction of functional dependencies
- 1984: ACM SIGMOD Contributions Award
- 1999: ACM SIGMOD Innovations Award
- 2002: IEEE John von Neumann Medal
- 2016: R.I.P.



Design Outline

- **Conceptual Model**
 - ER notation
 - Transformation to DB schema
- **Improving the Design**
 - **Normalization** (this week)
 - Performance tuning (weeks 7-9)



RDBMS

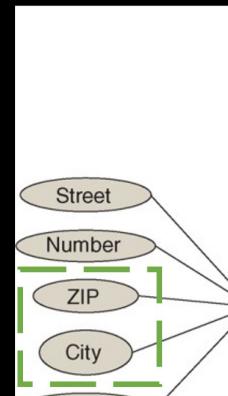
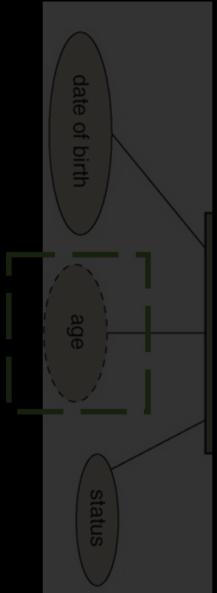
Normalization

Readings:

PDBM 3.0-3.3, 6.3-6.4

Derived Attributes in SQL DDL

- Not discussed in the PDBM book!
- Option 1: Create an attribute and maintain it
 - E.g. with a trigger, or regular update processes
- Option 2: Create a view that computes it
- Neither is very good!
- Sometimes there is a second kind of inter-attribute relationship
 - Here: ZIP → City
 - Called functional dependencies (FDs)
 - ER diagrams may miss such relationships!
 - We fix this with normalization (week 6)



Redundancy Issues

- Consider the table below Person(ID, Name, ZIP, City)
- What can go wrong during:
 - Insert? – what if new person lives in København V?
 - Update? – what if a municipality is renamed?
 - Delete? – what if Johan is deleted?
 - Also: Extra storage

Person

ID	Name	ZIP	City
1	Björn	2100	København Ø
2	Johan	2300	København S
3	Peter	2100	København Ø

Solution: Decompose the Relation

Person

ID	Name	ZIP
1	Björn	2100
2	Johan	2300
3	Peter	2100

ZIP

ZIP	City
2100	København Ø
2300	København S

css

```
SELECT P.ID, P.Name, P.ZIP, Z.City  
FROM Person P  
JOIN ZIP Z ON P.ZIP = Z.ZIP;
```

ID	Name	ZIP	City
1	Björn	2100	København Ø
2	Johan	2300	København S
3	Peter	2100	København Ø

How to Decompose?

- 1. Add new relation
- 2. Fill new relation
- 3. Alter old relation

sql

```
CREATE TABLE ZIP (
    ZIP INT PRIMARY KEY,
    City CHARACTER VARYING NOT NULL
);

INSERT INTO ZIP
SELECT DISTINCT ZIP, City
FROM Person;

ALTER TABLE Person ADD
FOREIGN KEY (ZIP) REFERENCES ZIP(ZIP);

ALTER TABLE Person DROP COLUMN City;
```

Practice: How to Decompose?

- 1. Add new relation(s)
- 2. Fill new relation(s)
- 3. Leave old relation unchanged!
 - Join of new relations – should give same data as old relation!

sql

```
-- CREATE and INSERT INTO ZIP as before  
-- No ALTER TABLE statements
```

```
CREATE TABLE NewPerson (  
    ID INT PRIMARY KEY,  
    Name CHARACTER VARYING NOT NULL,  
    ZIP INT NOT NULL REFERENCES ZIP(ZIP)  
);
```

```
INSERT INTO NewPerson  
SELECT ID, Name, ZIP  
FROM Person;
```

Redundancy Issues -- Revisited

- Consider the table below Person(ID, Name, ZIP, City)
- What can go wrong during:
 - Insert? – what if new person lives in København V?
 - Update? – what if a municipality is renamed?
 - Delete? – what if Johan is deleted?
 - Also: ~~Extra storage~~

Person

ID	Name	ZIP
1	Björn	2100
2	Johan	2300
3	Peter	2100

ZIP

ZIP	City
2100	København Ø
2300	København S

What Really Happened?

- Consider the table below Person(ID, Name, ZIP, City)

Person

ID	Name	ZIP	City
1	Björn	2100	København Ø
2	Johan	2300	København S
3	Peter	2100	København Ø

- Problem: Functional Dependency $\text{ZIP} \rightarrow \text{City}$**
 - If two records have the same ZIP value, they are guaranteed to have the same City value
 - ER does not capture this relationship easily
- Solution: Decomposition!**

Towards Normal Forms

- This table is in 2NF
 - Person: $\underline{ID} \rightarrow \text{Name}$
 - Person: $\underline{ID} \rightarrow \text{ZIP}$
 - Person: $\underline{ID} \rightarrow \text{City}$
 - Person: $\text{ZIP} \rightarrow \text{City}$
(transitive FD)

Person			
ID	Name	ZIP	City
1	Björn	2100	København Ø
2	Johan	2300	København S
3	Peter	2100	København Ø

- These tables are in BCNF
 - $\langle \text{key} \rangle \rightarrow \langle \text{attribute} \rangle$
 - Person: $\underline{ID} \rightarrow \text{Name}$
 - Person: $\underline{ID} \rightarrow \text{ZIP}$
 - ZIP: $\underline{\text{ZIP}} \rightarrow \text{City}$

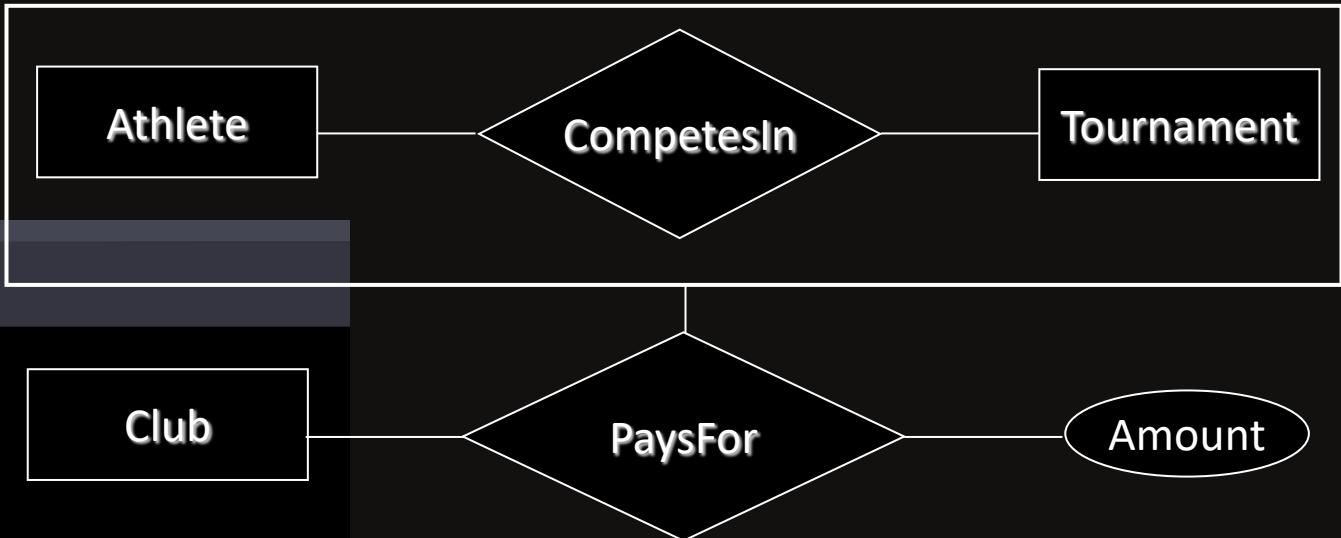
Person		
ID	Name	ZIP
1	Björn	2100
2	Johan	2300
3	Peter	2100

ZIP	
ZIP	City
2100	København Ø
2300	København S

- We like BCNF = no redundancy!
- Every attribute is dependent on the key, the whole key and nothing but the key!

Another Example: Mistake in Aggregation?

- Consider the PaysFor relationship from ER slides



sql

```
CREATE TABLE PaysFor (
    AID INTEGER,
    TID INTEGER,
    CID INTEGER REFERENCES Clubs,
    amount INTEGER NOT NULL,
    FOREIGN KEY (AID, TID) REFERENCES CompetesIn (AID, TID),
    PRIMARY KEY (AID, TID, CID)
);
```

Another Example: Add Data

- Perhaps AID and CID are related?

- What if $AID \rightarrow CID$ – athlete belongs to **one** club!
- Means that the key is in fact only AID, TID

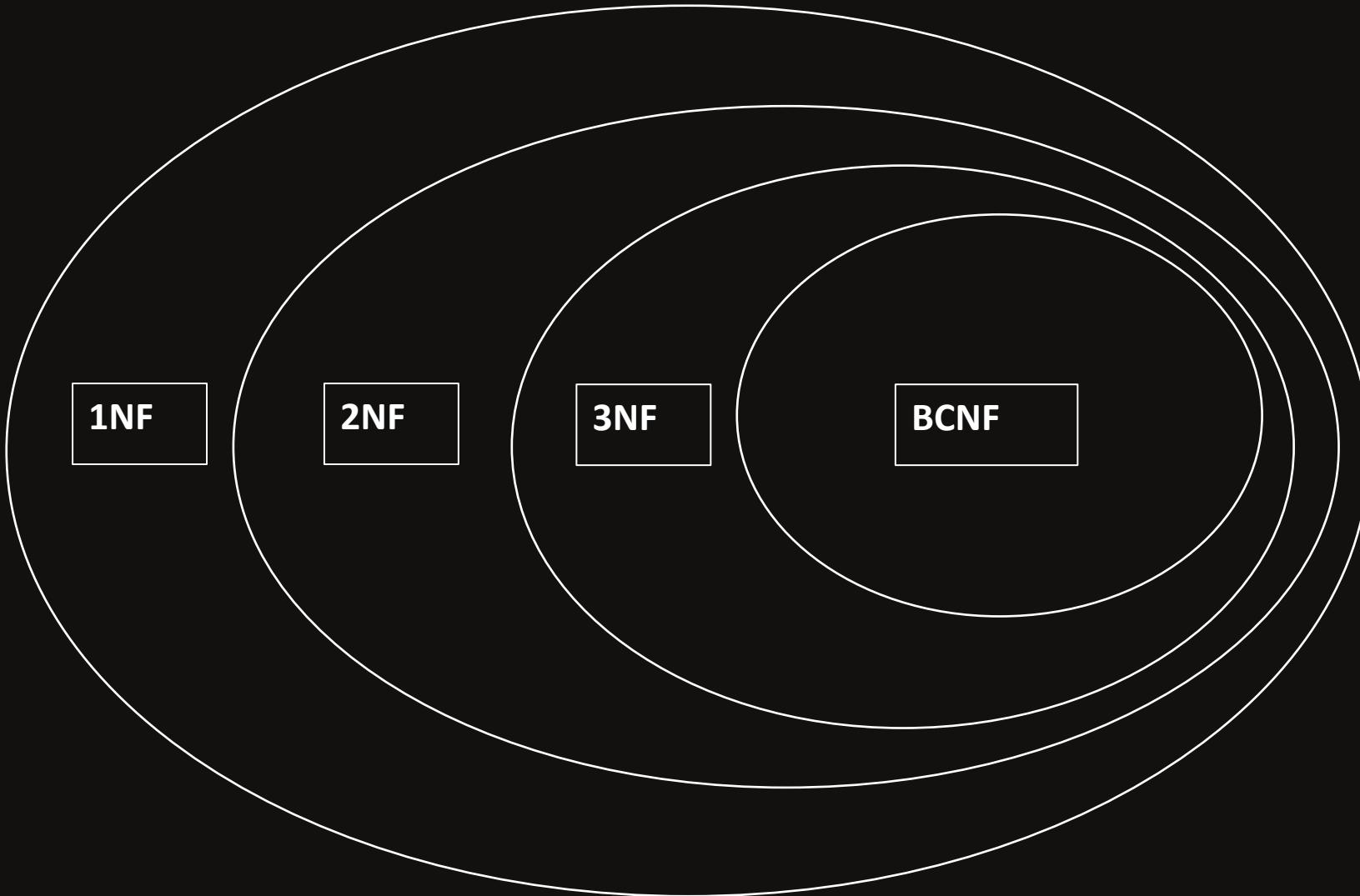
PaysFor

AID	TID	CID	amount
1	1	1	1000
1	2	1	5000
1	3	1	600
2	1	1	1000
3	2	2	1000

```
sql
CREATE TABLE PaysFor (
    AID INTEGER,
    TID INTEGER,
    CID INTEGER REFERENCES Clubs,
    amount INTEGER NOT NULL,
    FOREIGN KEY (AID, TID) REFERENCES CompetesIn (AID, TID),
    PRIMARY KEY (AID, TID, CID)
);
```

- This table is in 1NF
 - AID, TID \rightarrow Amount
 - AID \rightarrow CID

Normal Forms



Third Normal Form = 3NF

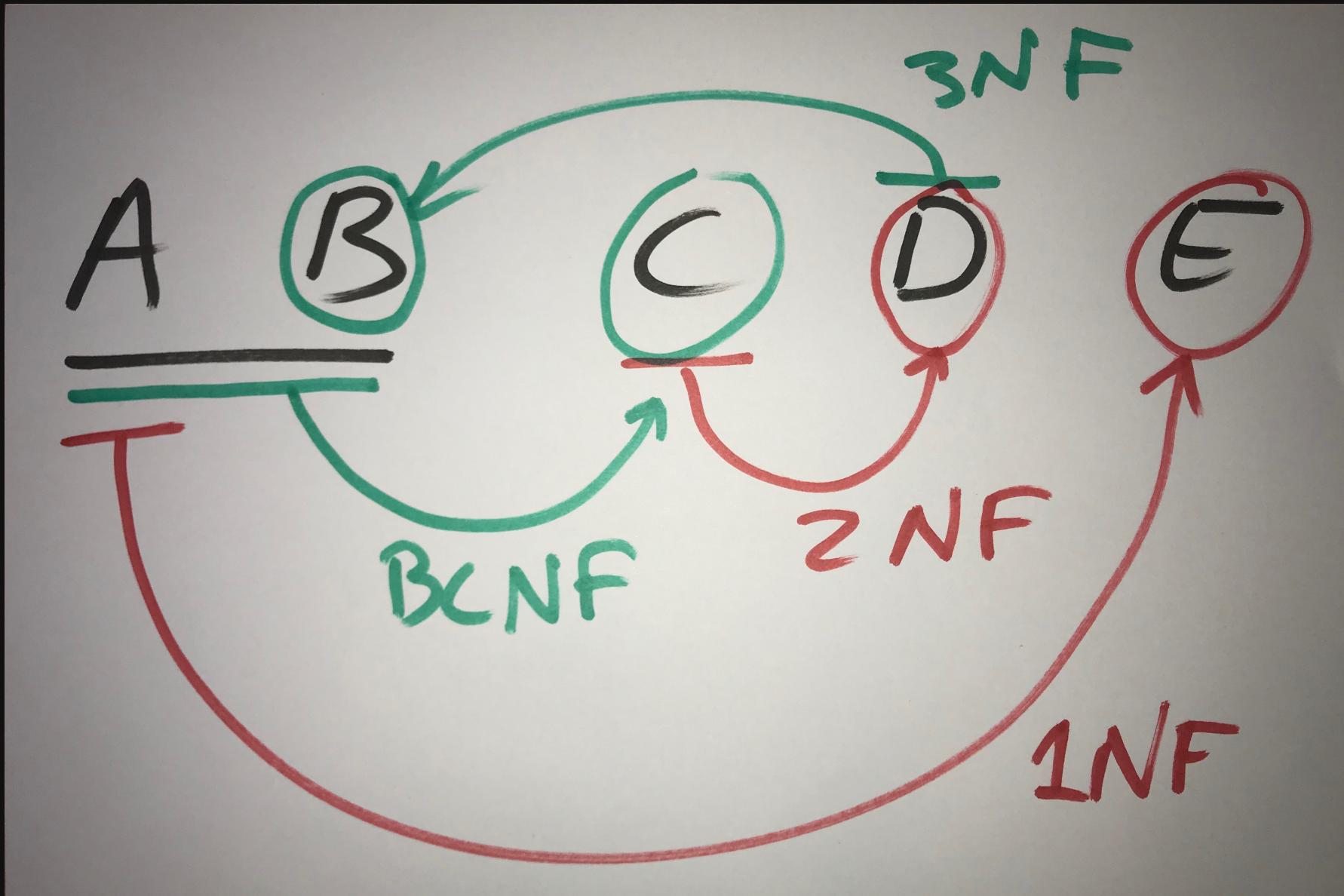
- We saw 1NF, 2NF and BCNF (= πNF, 3.5NF)
- What is 3NF?

- Prototype: $R(A, B, C), C \rightarrow A$
- Example from Iceland:
 - Boats(Area, Number, ZIP, ...)
 - Area = TH, PH, AK, ...
 - ZIP → Area
 - This is 3NF

- **We do NOT normalise this!**
 - We DO note that (Number, ZIP) is also a key (UNIQUE)



Visualizing FDs and Normal Forms



Can We Get a Simple Decomposition Algorithm?



Preview: Simple Decomposition Algorithm!

- 1. Find all important functional dependencies (FDs)
2. While FD < 3NF exists:
Decompose!

- Will result in BCNF or 3NF
 - No way from 3NF to BCNF ...
 - ... except by losing a dependency

Flash-Back to Keys I

- A candidate key for a relation is a set K of its attributes that satisfy:
 - Uniqueness: The values of the attribute(s) in K uniquely identify a tuple
 - Minimality: The uniqueness property goes away if we remove any attribute from K
- If only uniqueness is satisfied the attributes are said to form a superkey
- Example:
 - Person(ID, CPR, Name, ZIP)
 - (ID) is a candidate key
 - (CPR) is a candidate key
 - (CPR, Name) is a superkey
 - (ZIP, Name) is not a key

Flash-Back to Keys II

- **Important: Candidate key is defined with respect to what data can legally occur, not with respect to any particular instance of the relation**
 - Non-unique attributes are not keys
 - Unique attributes MAY be keys
- **One candidate key is selected as the primary key**
 - There could be several candidate keys to choose from
 - Example: Student(***id***, ***cpr***, ***exam_no***, ***email***, *name*)
 - *For normalization, it is irrelevant which key is chosen as primary key*

Keys and Functional Dependencies

- Key constraint is a special kind of functional dependency: all attributes of relation occur on the right-hand side of the FD
 - $ID \rightarrow ID, Name, ZIP$
OR
 - $ID \rightarrow ID, ID \rightarrow Name, ID \sqcap ZIP$
OR
 - $ID \rightarrow Name, ID \rightarrow ZIP$ (since $ID \rightarrow ID$ is clearly true)
- Keys are unique:
 - The left side cannot have the same values
 - ... but other attributes can

Functional Dependencies ($\mathcal{FD}s$)

- **Definition:**

- A functional dependency (FD) on a relation schema R is a constraint $X \rightarrow Y$, where X and Y are subsets of attributes of R
- A FD $X \rightarrow Y$ is satisfied in an instance r of R if for every pair of tuples, t and s:
if t and s agree on all attributes in X then they must agree on all attributes in Y
- Confused? Think about $X = \text{ZIP} \rightarrow Y = \text{City}$

Note

Intuitively: If X and Y were the only attributes in a relation, then X would be a key!

Two (Un-)Important Types of FDs

- FDs with a superkey on the left are **unavoidable**
 - If A is a candidate key for a relation then clearly $A \rightarrow B$ for any attribute B
 - Similarly if $\{A_1, A_2\}$ forms a superkey we have $A_1A_2 \rightarrow B$ for any B, etc
- FDs like $X \rightarrow A$ are **trivial** if $A \in X$
 - Therefore $X \rightarrow Y$ is trivial if $Y \subseteq X$
- They are (un-)important because they do not require decomposition!

Redundant FDs

- If we already know that $ID \rightarrow ZIP$ **is** ID, $Name \rightarrow ZIP$ useful?
- If we already know that $ID \rightarrow ZIP$ and $ZIP \rightarrow City$ **is** ID \rightarrow City useful?
- These are examples of redundant FDs
 - Can be computed from others
 - Do not require decomposition!
- Can be formally defined and computed
 - ... but you'll know them when you see them!

What is Decomposition?

- Consider relation R and (important) functional dependency $X \rightarrow Y$ that violates 3NF/BCNF
 - Decompose R into R1 and R2 where
 - $R_1 = R - Y$ (everything but Y = the right side)
 - $R_2 = XY$ (the whole FD = both left and right side)
- This has the following nice properties
 - R_2 is (normally) in BCNF
 - Joining R_1 and R_2 (with = on all X attributes) yields R
- Example: Person(ID, Name, ZIP, City), $ZIP \rightarrow City$
 - $X = ZIP$, $Y = City$
 - $R - Y = Person(ID, Name, ZIP)$
 - $XY = ZIP(ZIP, City)$

Our Simple Decomposition Algorithm!

- 1. **Find all the FDs**
Remove trivial, unavoidable, redundant FDs
- 2. **While FD < 3NF exists:**
Decompose!
- **Will result in BCNF or 3NF**
 - No way from 3NF to BCNF ...
 - ... except by losing a dependency
 - ... so we don't decompose 3NF!!!

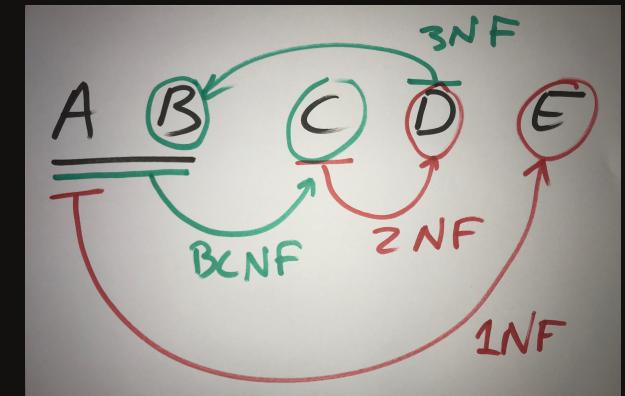
Example: Decomposition

- $R = \underline{ABCD}$
- $AB \rightarrow CD$
- $C \rightarrow D$

- New Table: CD
- Old Table: ABC

- Both in BCNF
- What happened to $AB \rightarrow D$?

$$\begin{aligned} X &\rightarrow Y \\ R_1 &= R - Y \\ R_2 &= XY \end{aligned}$$

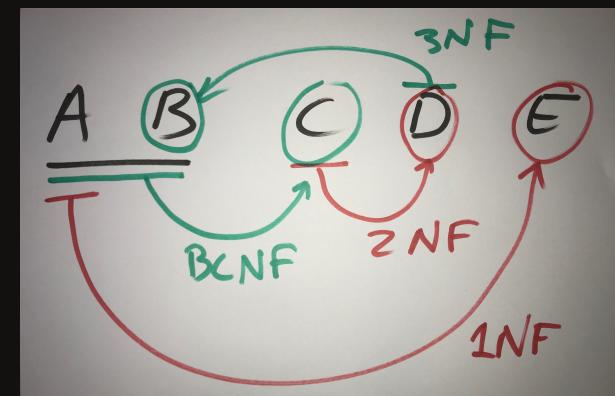


Example: Decomposition

- $R = \underline{ABCD}$ ($AB \rightarrow CD$)
- $A \rightarrow D$
 - A = personID, B = projectID, C = start date, D = name of person

$X \rightarrow Y$
 $R1 = R - Y$
 $R2 = XY$

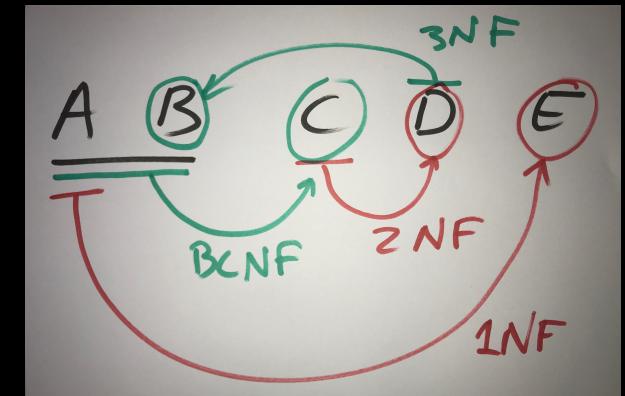
- New Table: AD
- Old Table: ABC
- Both in BCNF
- What happened to $AB \rightarrow D$?



Example: Decomposition

- $R = \underline{ABCD}$
- $A \rightarrow C$
- $A \rightarrow D$
 - $A \rightarrow CD$
- New Table: ACD // Merge tables with identical keys
- Old Table: AB
- Both in BCNF

$X \rightarrow Y$
 $R1 = R - Y$
 $R2 = XY$

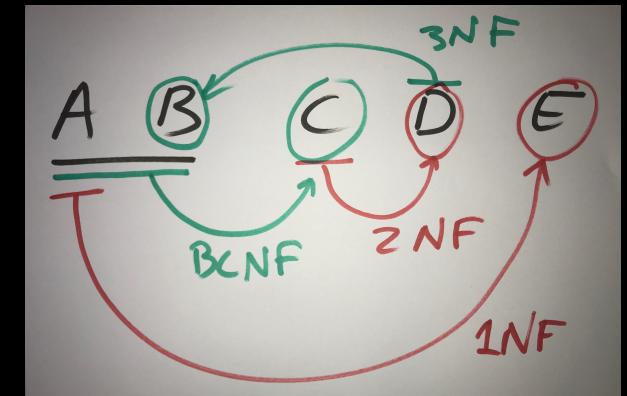


Example: Decomposition

- $R = \underline{ABCD}$
- $D \rightarrow A$

$X \rightarrow Y$
 $R1 = R - Y$
 $R2 = XY$

- R is in 3NF
- No decomposition into BCNF

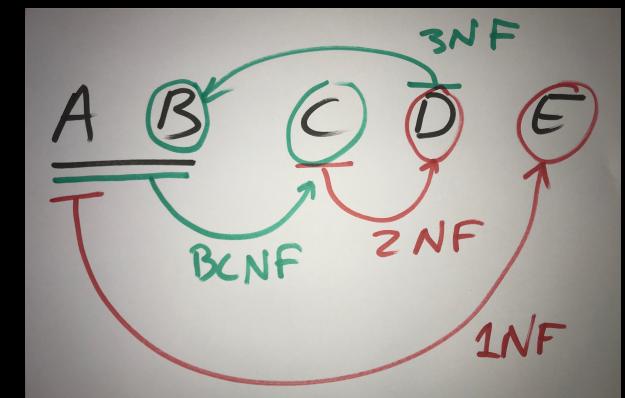


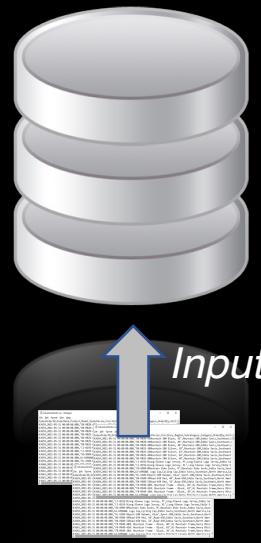
Example: Decomposition

- $R = \underline{ABCD}$
- $A \rightarrow C, C \rightarrow D$
- Which one do we use first? ... and why?

$X \rightarrow Y$
 $R1 = R - Y$
 $R2 = XY$

- New Table: CD
 - New Table: AC
 - Old Table: AB
-
- All three in BCNF
 - What happened to $AB \rightarrow D$





RDBMS

How Do We Find Keys?

Finding Keys

- **Is attribute A a key to R?**
 - What does it determine (directly and indirectly)
 - Recursively add “right sides” of FDs to A until done...
 - If A determines R: Yes
- **Are attribute set X a key to R?**
 - What do they together determine (directly and indirectly)
 - Recursively add “right sides” of FDs to X until done...
 - If X determines R: Maybe
 - If no subset of X is a key: Yes
- **Finding all keys of R:**

In theory: Examine all attribute subsets of R
In practice: Focus on left sides of FDs, avoid supersets

Finding Keys: Examples

- Consider the table Person(ID, Name, ZIP, City)

Person

ID	Name	ZIP	City
1	Björn	2100	København Ø
2	Johan	2300	København S
3	Peter	2100	København Ø

- FDs: $ID \rightarrow name$, $ID \rightarrow ZIP$, $ZIP \rightarrow City$
 - Is City key?
No → not on left side of any FD
 - Is ID key?
Yes → determines Name, ZIP, City
 - Is ZIP key?
No → only determines ZIP, City
 - Is $(ID, name)$ key?
No → superset of key

Two (or more) Keys and BCNF

- Consider this Person relation
 - With two keys!

Person

ID	CPR	Name	ZIP
1	0123456789	Björn	2100
2	9876543210	Johan	2300
3	1122334455	Peter	2100

```
sql
CREATE TABLE Person (
    ID INT PRIMARY KEY,
    CPR CHAR(10) NOT NULL UNIQUE,
    Name CHARACTER VARYING NOT NULL,
    ZIP INT NOT NULL REFERENCES ZIP(ZIP)
);
```

Two (or more) Keys and BCNF

- Consider this Person relation

Person

ID	CPR	Name	ZIP
1	0123456789	Björn	2100
2	9876543210	Johan	2300
3	1122334455	Peter	2100

- What are the FDs?
 - $\underline{ID} \rightarrow \underline{CPR}$, $\underline{ID} \rightarrow \text{Name}$, $\underline{ID} \rightarrow \text{ZIP}$
 - $\underline{CPR} \rightarrow \underline{ID}$, $\underline{CPR} \rightarrow \text{Name}$, $\underline{CPR} \rightarrow \text{ZIP}$
 - All are: <key> \rightarrow <something>
- The table is in BCNF – there is no redundancy!

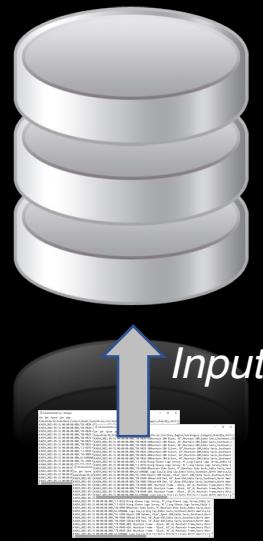
$3NF$ and Keys

- Prototype: $R(\underline{A}, \underline{B}, C), C \rightarrow A$
 - AB is a key
 - CB is also a key!
 - 3NF always has (at least) two keys!
- Example from Iceland:
 - Boats(Area, Number, ZIP, ...)
 - ZIP → Area
- We do NOT normalise this!
 - We DO note that Number, ZIP is also a key (UNIQUE)



Example: Normalization

- **Advices**(Teacher, Student, Dept, Program, Course)
 - Teacher → Dept
 - Course → Program
- **Outcome:**
 - Faculty(Teacher, Dept)
 - Catalog(Course, Program)
 - Advices(Teacher, Student, ~~Dept, Program~~, Course)



RDBMS

Raw Data

How do We Find Functional Dependencies?

Discovering FDs: Inspection Method

- How can we find FDs?

Person

ID	Name	ZIP	City
1	Björn	2100	København Ø
2	Johan	2300	København S
3	Peter	2100	København Ø

- Run a “thought experiments”:
 - Assume that an attribute is a key of a sub-relation
 - Consider which attributes could be in that relation?
- Study application design requirements:
 - Ex: Each vendor can only sell one part to each project
 - vendor project → part

Discovering FDs: SQL Method

- **Study relation instances (when available)**
 - + Check application design requirements
 - + Check reality

Person

ID	Name	ZIP	City
1	Björn	2100	København Ø
2	Johan	2300	København S
3	Peter	2100	København Ø

What is the SQL method?

- Assume that $\text{ZIP} \rightarrow \text{City}$ holds:

- For each ZIP value, how many different City values?

```
sql
SELECT 'Person: ZIP --> City' AS FD,
       CASE WHEN COUNT(*)=0 THEN 'MAY HOLD'
             ELSE 'does not hold' END AS VALIDITY
FROM (
    SELECT P.ZIP
    FROM Person P
    GROUP BY P.ZIP
    HAVING COUNT(DISTINCT P.City) > 1
) X;
```

Person			
ID	Name	ZIP	City
1	Björn	2100	København Ø
2	Johan	2300	København S
3	Peter	2100	København Ø

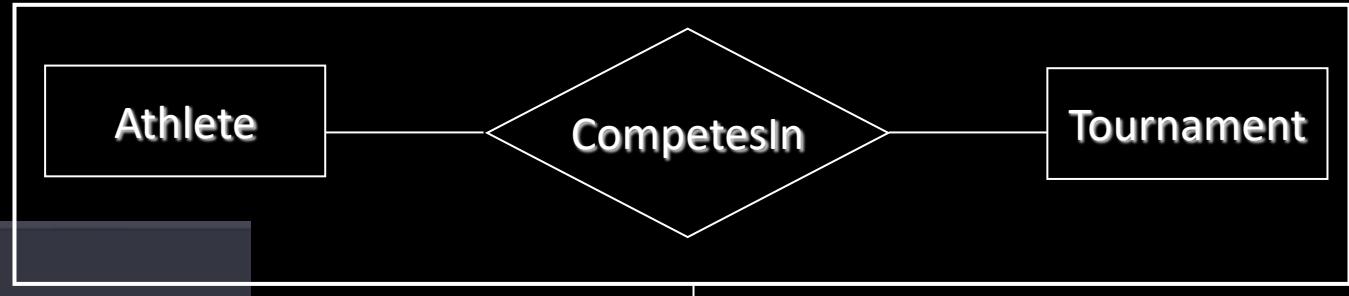
Discovering FDs: SQL Method

- **Study relation instances (when available)**
 - Can make a script to test all combinations, e.g.:
 - Use Java to write SQL queries into text file
 - Run the text file using psql
 - **Remember: Script can only say MAY HOLD!**
 - Consider City → ZIP
- + Check application design requirements
+ Check reality

Person			
ID	Name	ZIP	City
1	Björn	2100	København Ø
2	Johan	2300	København S
3	Peter	2100	København Ø

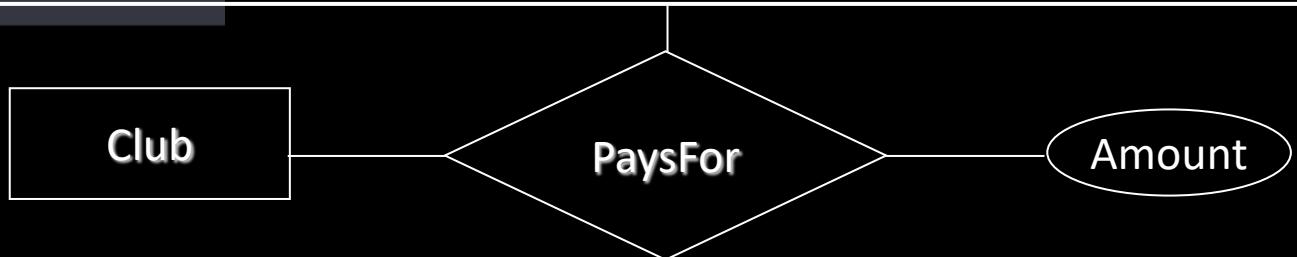
Another Example Revisited (slides 15-16)

- Consider the PaysFor relationship from ER slides



sql

```
CREATE TABLE PaysFor (
    AID INTEGER,
    TID INTEGER,
    CID INTEGER REFERENCES Clubs,
    amount INTEGER NOT NULL,
    FOREIGN KEY (AID, TID) REFERENCES CompetesIn (AID, TID),
    PRIMARY KEY (AID, TID, CID)
);
```



Another Example: Add Data

- Perhaps AID and CID are related?

- What if AID → CID – athlete belongs to **one** club!
- Means that the key is in fact only AID, TID

PaysFor

AID	TID	CID	amount
1	1	1	1000
1	2	1	5000
1	3	1	600
2	1	1	1000
3	2	2	1000

```
sql
CREATE TABLE PaysFor (
    AID INTEGER,
    TID INTEGER,
    CID INTEGER REFERENCES Clubs,
    amount INTEGER NOT NULL,
    FOREIGN KEY (AID, TID) REFERENCES CompetesIn (AID, TID),
    PRIMARY KEY (AID, TID, CID)
);
```

- This table is in 1NF
 - AID, TID → Amount
 - AID → CID

Another Example: Correct Tables

sql

```
CREATE TABLE AthleteClub (
    AID INTEGER PRIMARY KEY,
    CID INTEGER NOT NULL REFERENCES Clubs
);

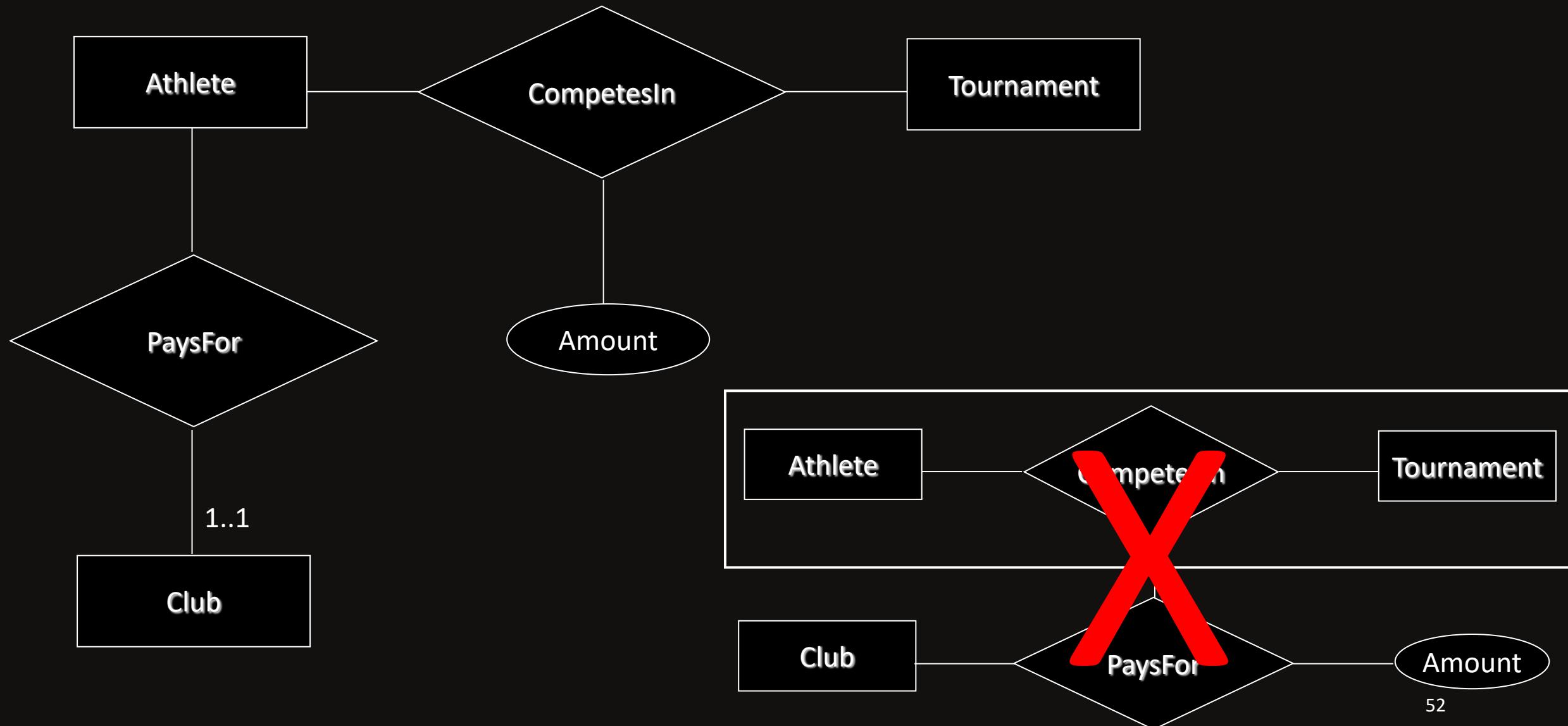
CREATE TABLE PaysFor (
    AID INTEGER,
    TID INTEGER,
    CID INTEGER REFERENCES Clubs,
    amount INTEGER NOT NULL,
    FOREIGN KEY (AID, TID) REFERENCES CompetesIn (AID, TID),
    PRIMARY KEY (AID, TID, CID)
);
```

Another Example: Really Correct Tables

```
sql  
  
CREATE TABLE Athlete (  
    AID INTEGER PRIMARY KEY,  
    ... -- Other columns here  
    CID INTEGER NOT NULL REFERENCES Clubs  
);
```

```
CREATE TABLE CompetesIn (  
    AID INTEGER REFERENCES Athlete,  
    TID INTEGER REFERENCES Tournament,  
    amount INTEGER NULL,  
    PRIMARY KEY (AID, TID)  
);
```

Another Example: Corrected ER Diagram





Takeaways

- **Functional Dependencies**
 - With flash-back to keys
 - Unavoidable, trivial, and redundant FDs
 - How to detect potential FDs?
- **Normal forms**
 - Focus on BCNF and 3NF
 - Always decompose 1NF and 2NF!
 - Never decompose 3NF!
- **Normalization process**
 - Decomposition of relations to BCNF (or 3NF)

Note

Contrary to popular belief, practical normalisation is not hard!



What is next?

- **Next Lecture:**
 - Storage Hierarchy
 - Multicore Processing
 - OS
- **Lecturer:**
 - Pınar Tözün

introduction to database systems storage, multicores, OS

Pınar Tözün
March 22, 2023

some slides are inspired by
Patterns in Data Management and Databases on Modern Hardware books &
Anastasia Ailamaki's Intro to Database Systems class at EPFL

agenda

- storage hierarchy
- hardware parallelism on multicores
- operating systems

why do we need to know these?

**impacts how we design database systems & leads
to better optimizations for faster data processing**

systems stack overview

application



e.g., online shopping page, database system, code to read/write a file, etc.

operating system

e.g., linux, windows, etc.



hardware

e.g., intel server, disks, etc.

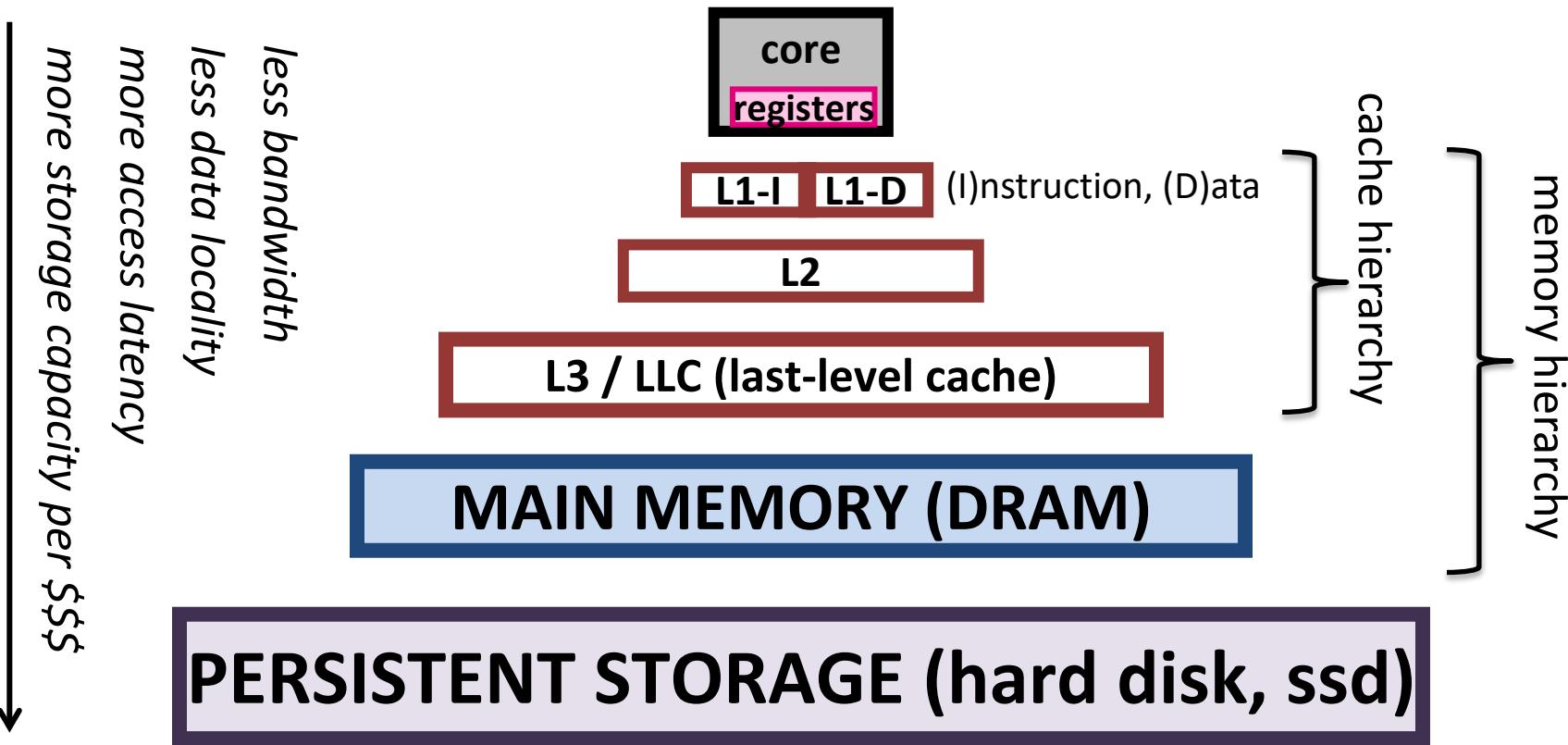


agenda

- **storage hierarchy**
- hardware parallelism on multicores
- operating systems

(typical) storage hierarchy

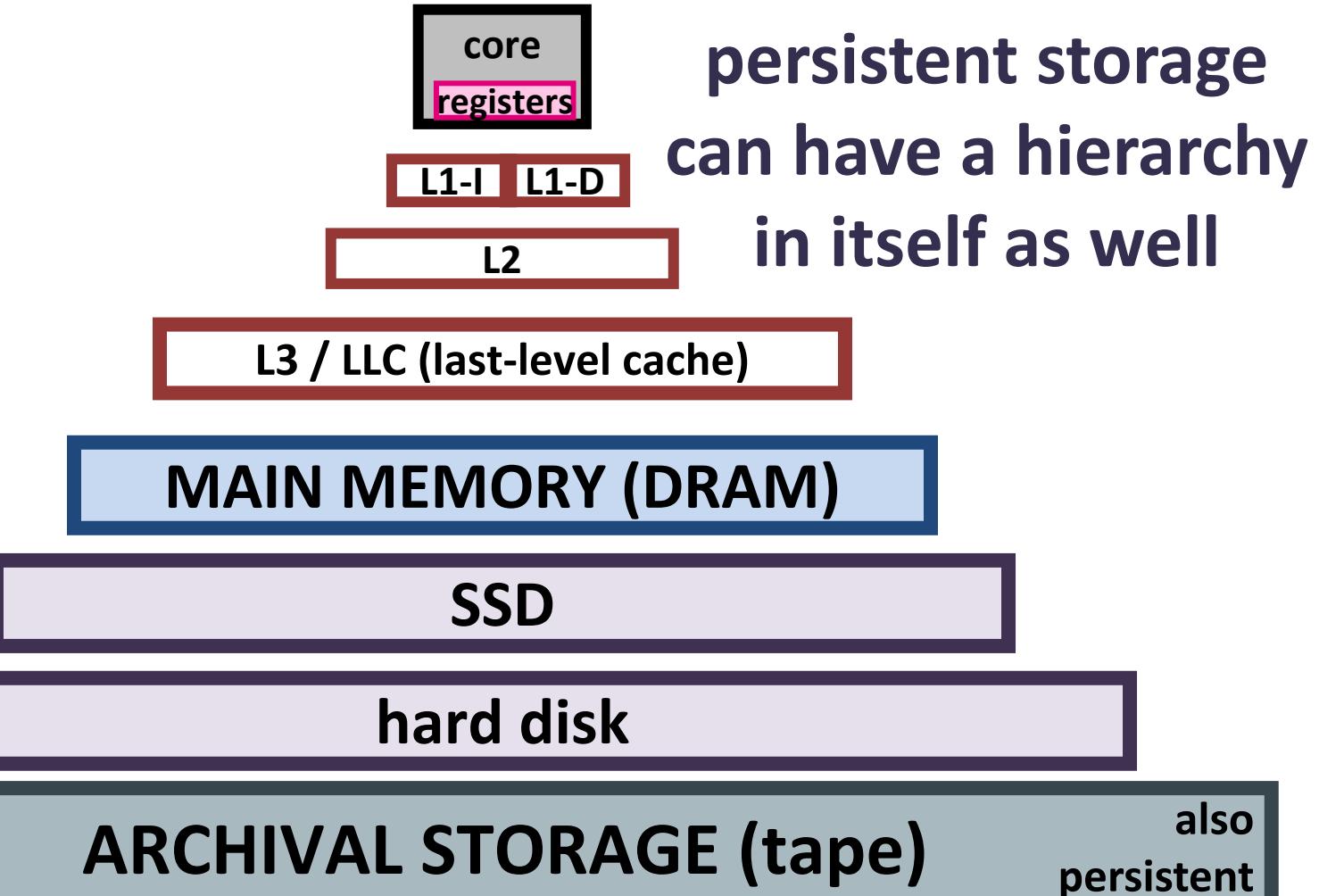
disclaimer: memory hierarchy
is based on Intel Xeons' here



ARCHIVAL STORAGE (tape)

also
persistent

(typical) storage hierarchy



(typical) storage hierarchy

distributed setting
(e.g., cluster of
machines)



persistent storage
can have a hierarchy
in itself as well

L3 / LLC (last-level cache)

MAIN MEMORY (DRAM)

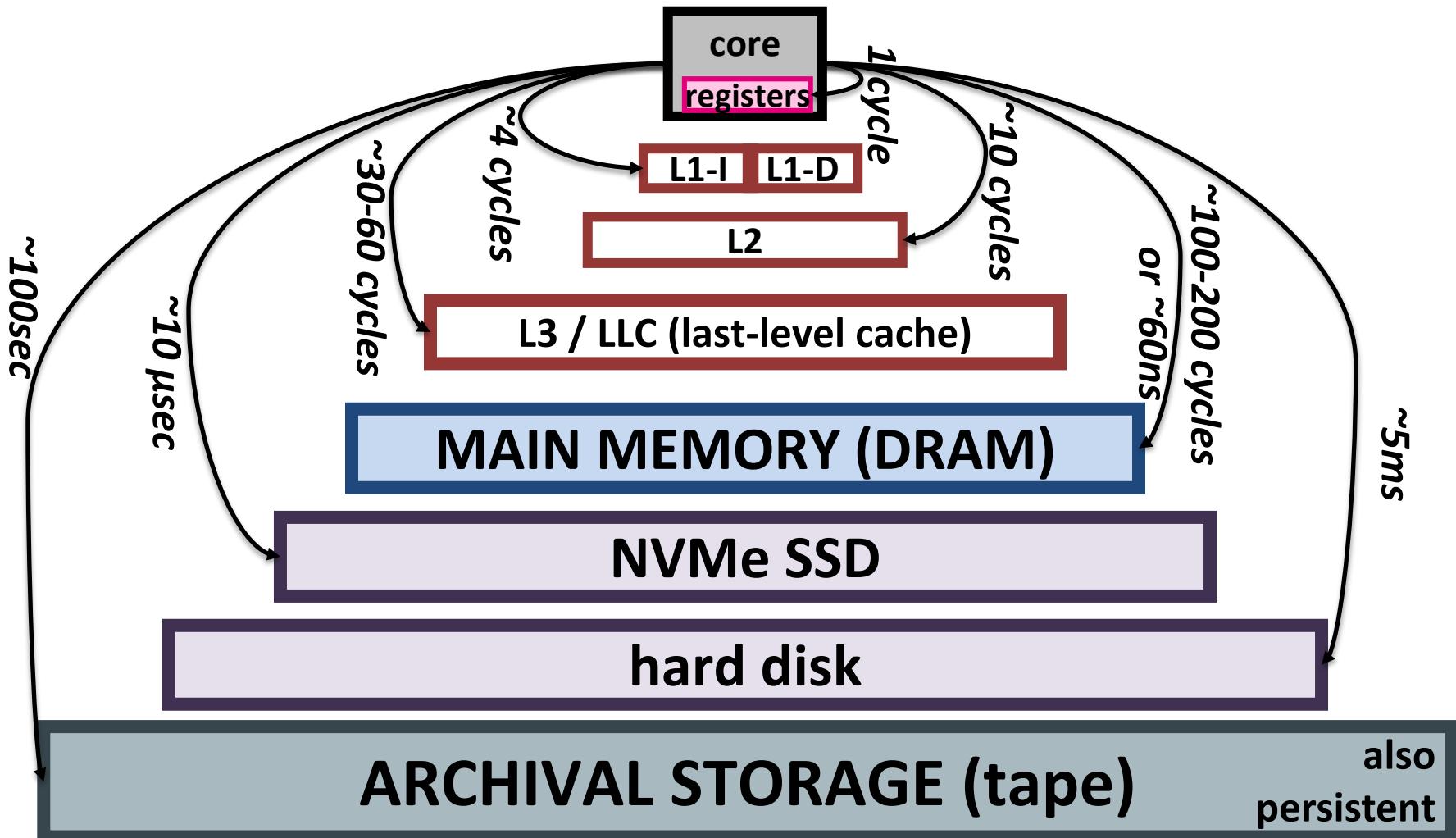
local disk

remote disk

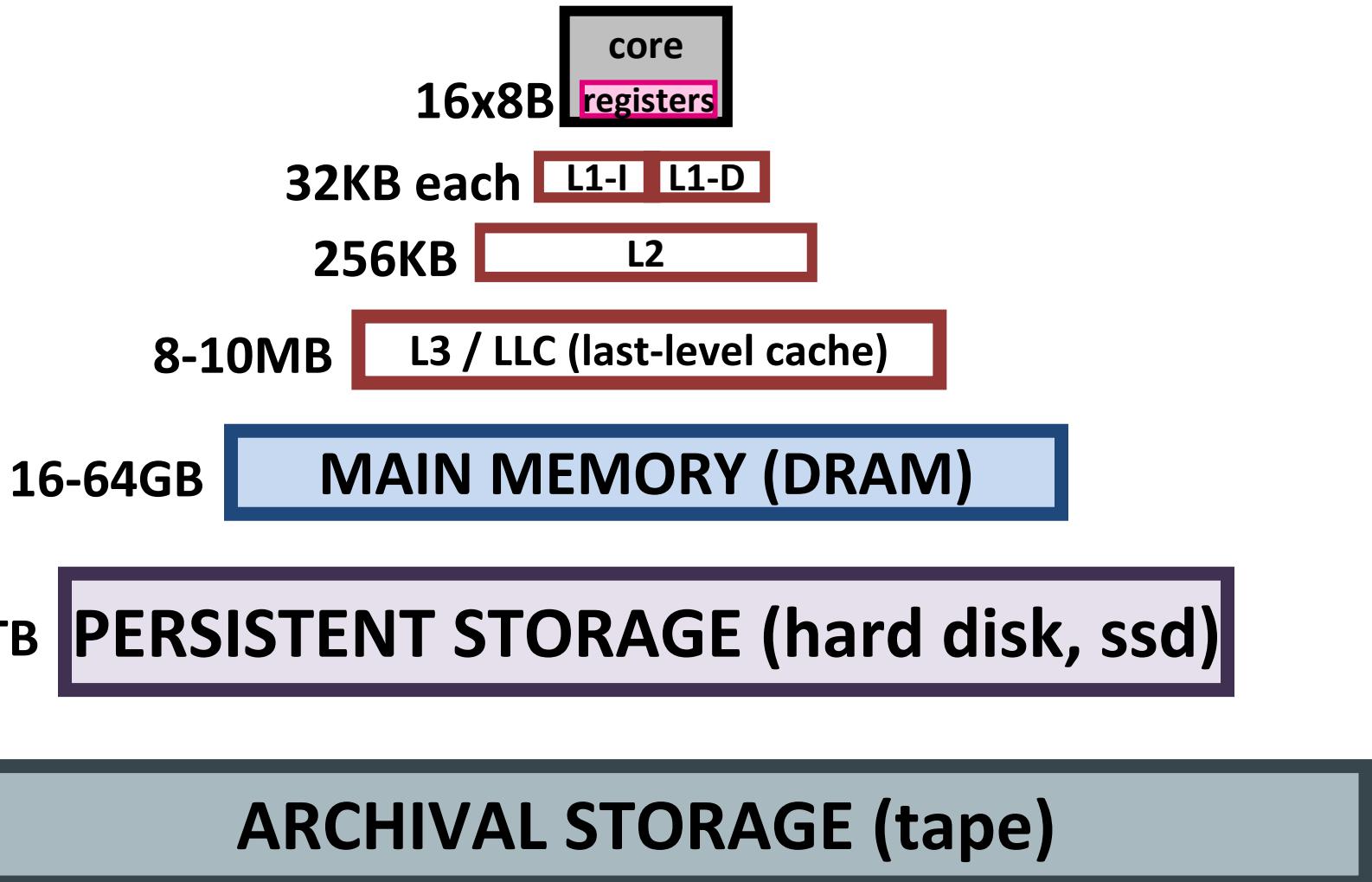
ARCHIVAL STORAGE (tape)

also
persistent

(typical) storage hierarchy – access latency



(typical) storage hierarchy – capacity

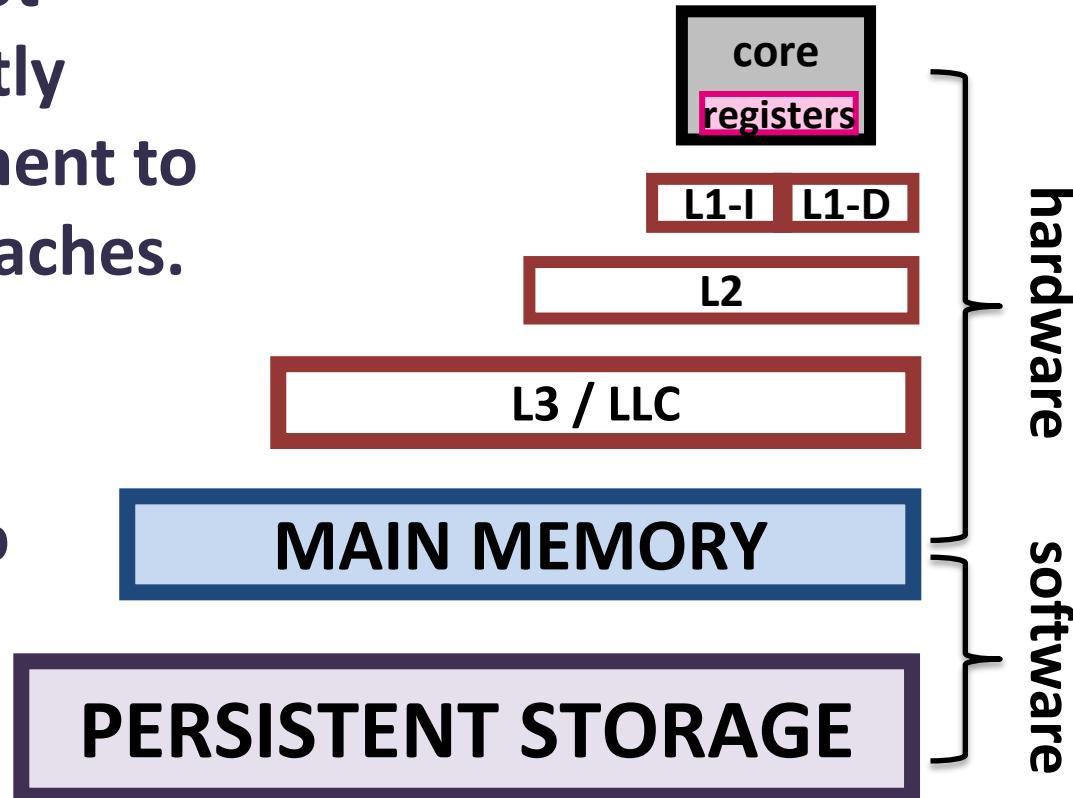


(typical) storage hierarchy – management

means that we do not write code to explicitly manage data movement to registers, L1, L2, L3 caches.

but we do this for moving data from/to persistent storage.

typically managed by ...



storage hierarchy

goal is to increase data locality for cores

to reduce access latency for frequently accessed data

higher levels cache data from lower levels inclusivity

- lower levels include all the data from higher levels (usually)
- most hardware vendors build inclusive cache hierarchy
- software controlled caching can be more complex
 - e.g., some database systems don't persist indexes on disk, they keep them in main memory

data replacement when no space left

replacement policy can be a crucial system optimization

tape

cheap way of storing voluminous data

only allows *sequential access*

does not allow *random access*

only used for archival storage today

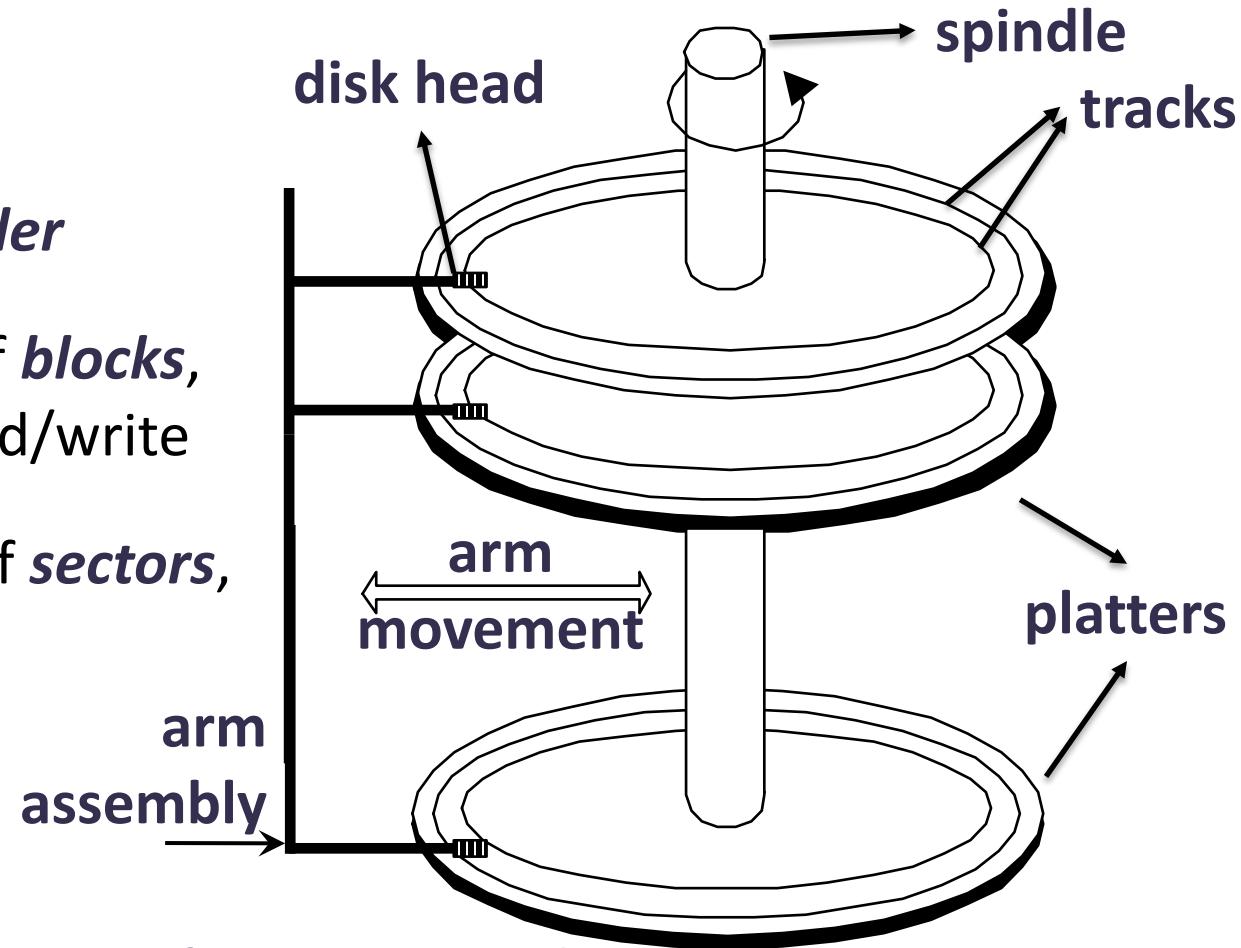


hard disk

tracks with the same diameter make a *cylinder*

tracks are composed of *blocks*, which is the unit of read/write

blocks are composed of *sectors*, which are of fixed-size



placement of data on disk is a crucial concern for access latency!

access latency on hard disk

also called I/O (input/output) latency

= seek time + rotational delay + transfer time

seek time

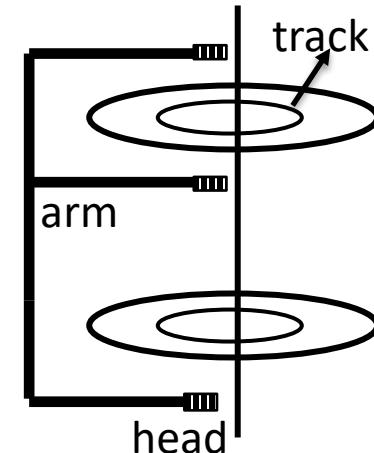
moving disk arm to the right track – (~1-20 ms)

rotational delay

reaching the desired block on the track – (~0-10 ms)

transfer time

reading/writing the desired data on the block – (~<1ms for 4KB)
i.e., disk head rotating over the block



for faster access →

minimize seek time & rotational delay!

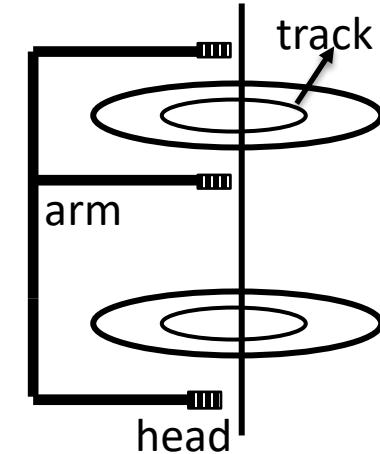
random vs. sequential access on hard disk

access latency for reaching a disk block = a =
seek time + rotational delay

reading a block = a + transfer time

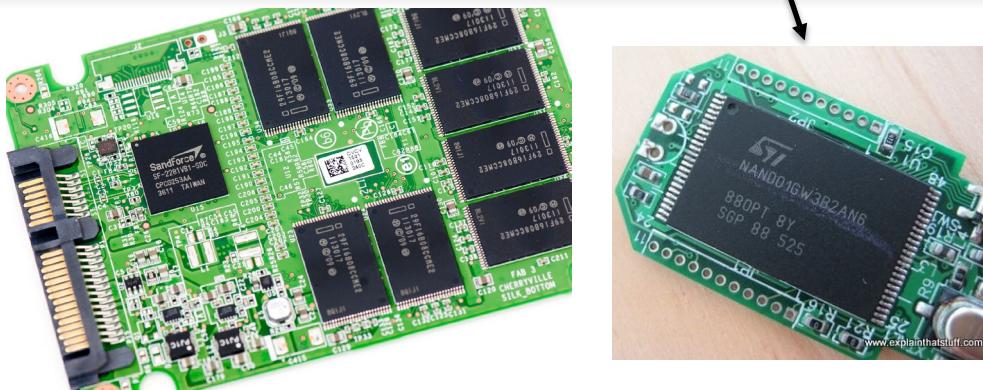
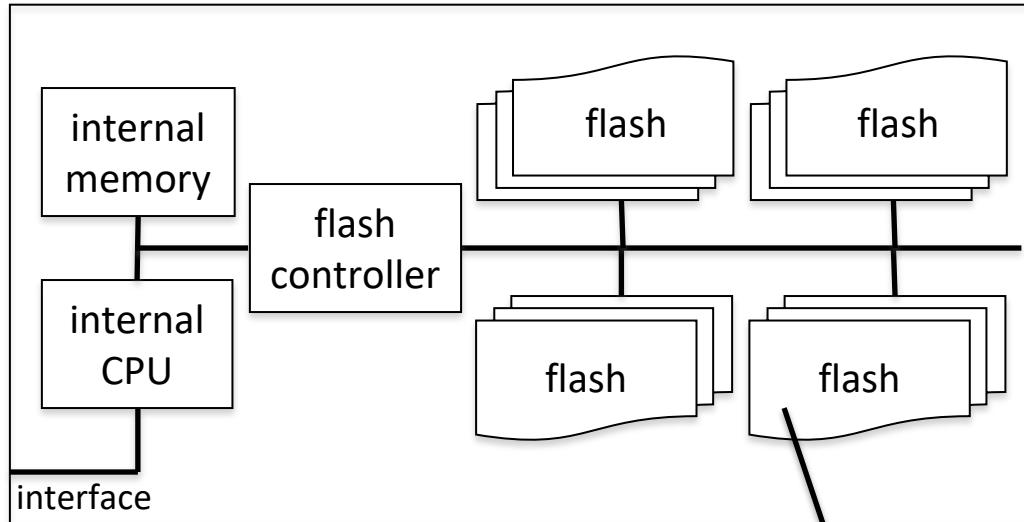
access latency for 100 random blocks =
 $100 \times (a + \text{transfer time})$

access latency for 100 sequential blocks =
 $a + 100 \times (\text{transfer time})$



sequential access is much faster than random access!
underlying principle for many optimizations in data systems

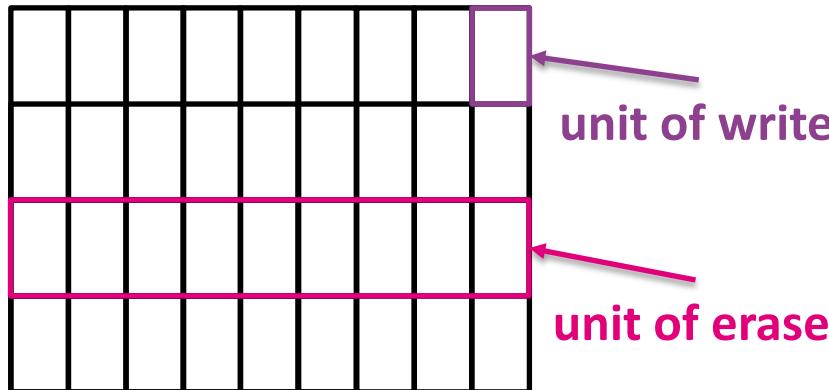
solid-state disk (SSD)



interconnected flash chips
efficient random access
internal parallelism
hard disk compatible API

**why not have as
drop-in replacement
for hard disks?**

solid-state disk (SSD)



can use it as drop-in replacement for hard disks,
but need to be smarter to more effectively exploit SSDs & not to burn money!

cannot override a unit before erasing it first

garbage collection – for not used blocks so we can rewrite them

write amplification = $\text{data physically written} / \text{data logically written} \geq 1$
writing data might cause rewrites & garbage collection

wear leveling – some cells/blocks die over time

unpredictable read/write latencies

if a request gets stuck after a write triggering garbage collection

random-access memory (RAM)

(almost) ***constant random-access latency*** wherever the data is

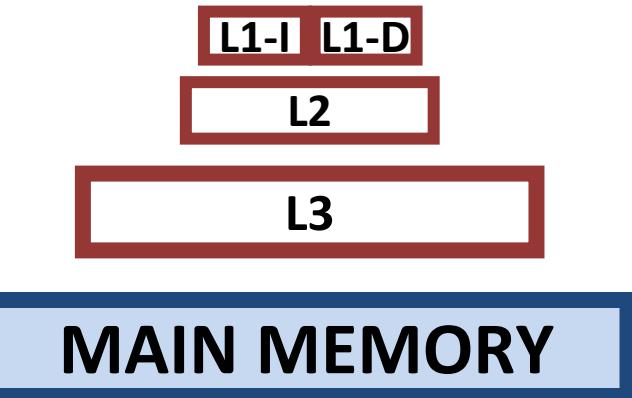
volatile*

will lose data once power is lost

sequential access is slightly faster still

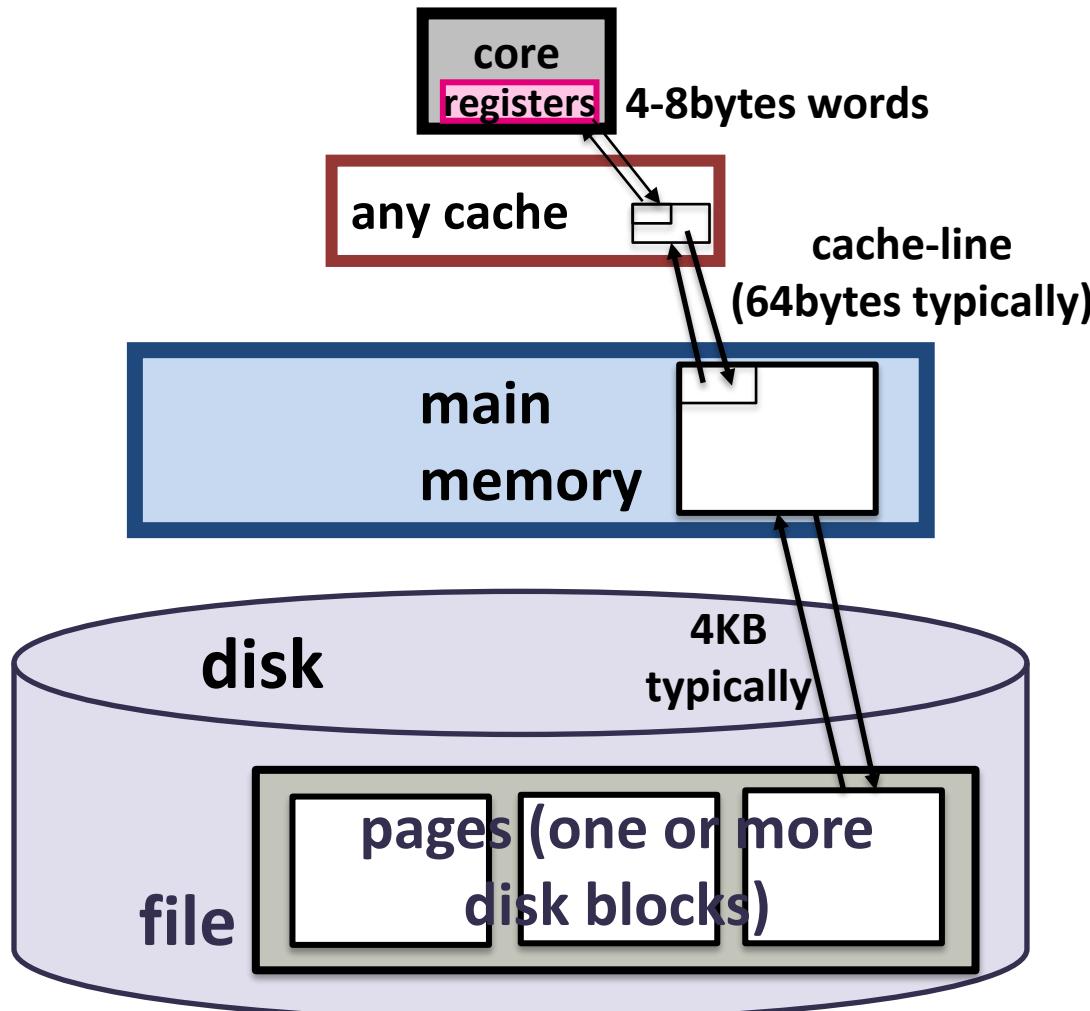
prefetching:

if you fetch a block from main-memory to caches,
hardware usually prefetches the adjacent block



*non-volatile/persistent memory is available, but not adopted in main-stream

movement of data in storage hierarchy



summary – storage hierarchy

Storage hierarchy is there to improve locality for frequently accessed data.

Different layers of the hierarchy have different characteristics & require different optimizations from the software side.

Sequential access is faster than random access for all layers, but especially for hard disks.

Data management systems have various optimizations to exploit the storage hierarchy the best possible way.

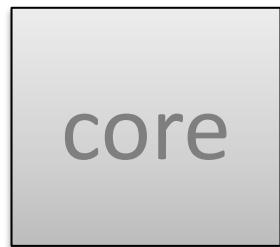
goal: minimize data access latency!

agenda

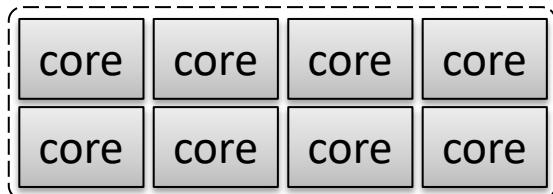
- storage hierarchy
- **hardware parallelism on multicores**
- operating systems

central processing unit (CPU) evolution

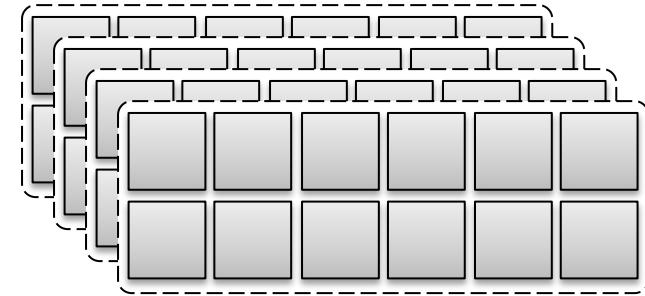
2005



single-core CPUs



multicore CPUs

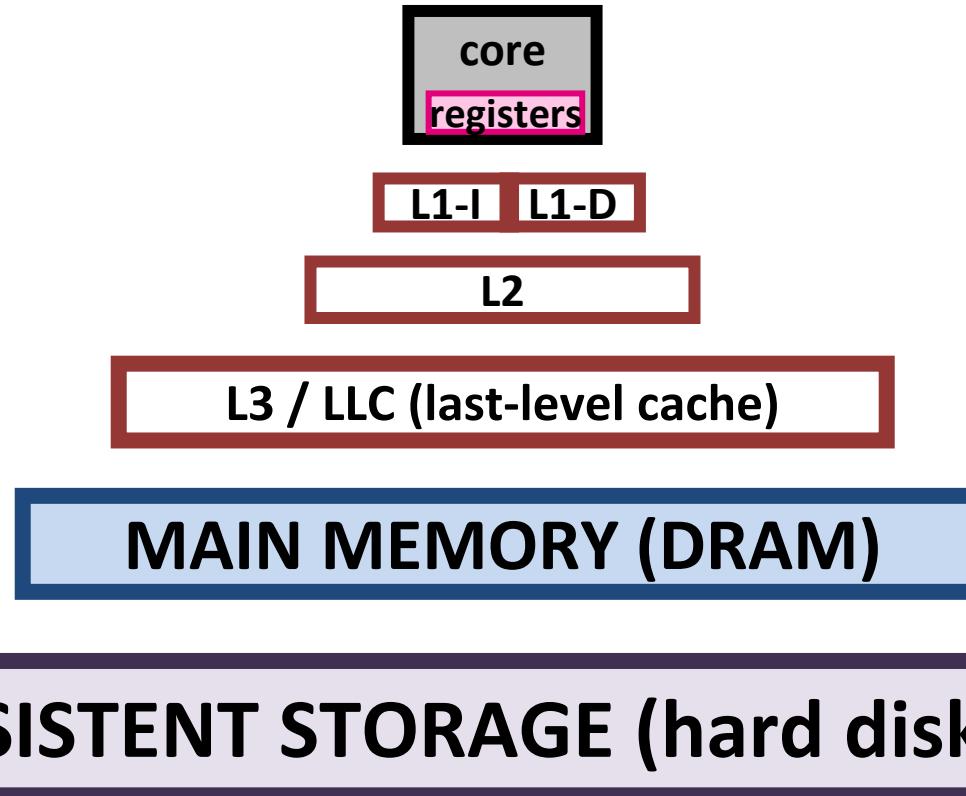


multisocket
multicore CPUs

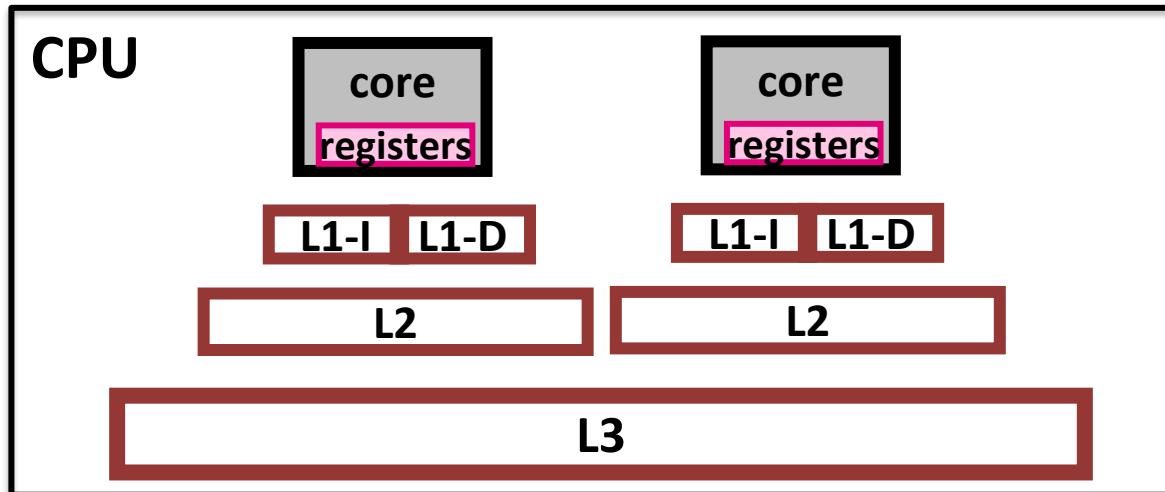
*faster & more-complex
cores over time*

*similar speed & complexity in a core,
more parallelism over time*

single-core storage hierarchy

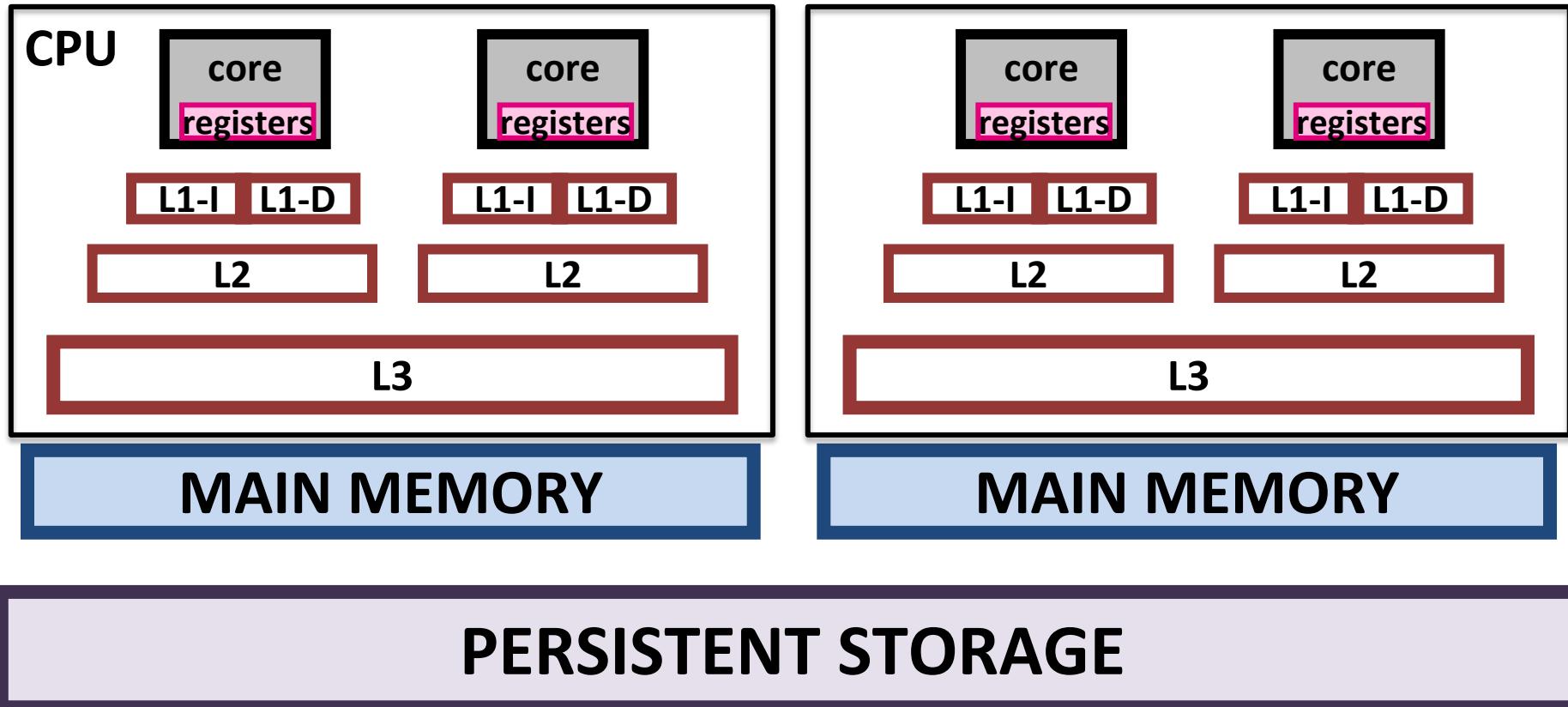


multicore storage hierarchy

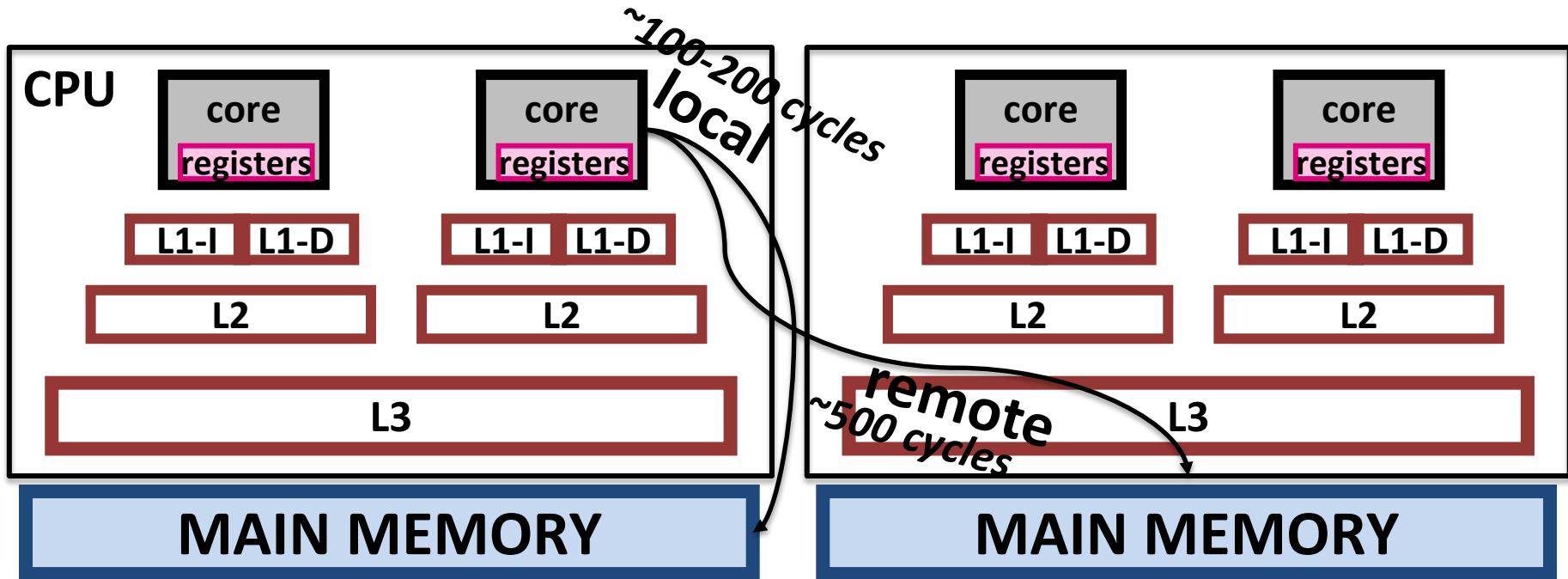


PERSISTENT STORAGE

multi-socket multicore storage hierarchy



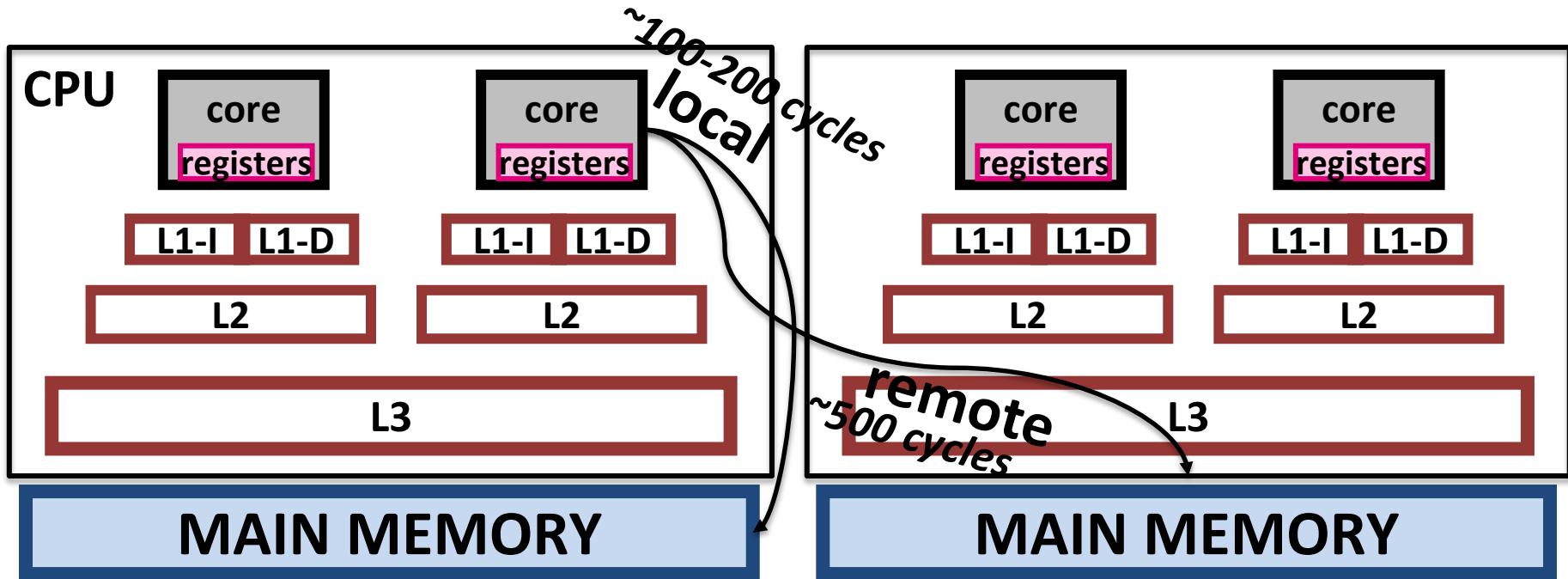
multi-socket multicore storage hierarchy



PERSISTENT STORAGE

local memory access is faster than remote one!

multi-socket multicore storage hierarchy

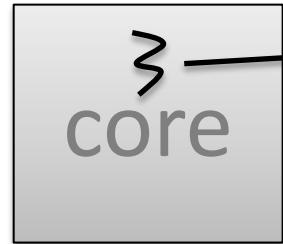


PERSISTENT STORAGE

also called NUMA, non-uniform memory access

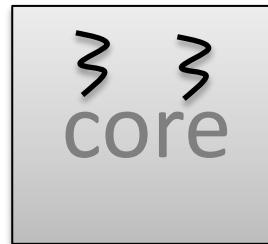
types of hardware parallelism

implicit parallelism

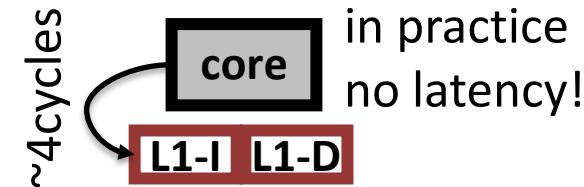


thread
what runs
your programs

instruction & data parallelism
hardware does this automatically



multithreading
threads share
execution cycles
on the same core



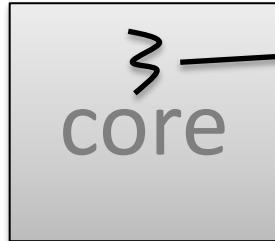
MAIN MEMORY

goal: minimize stall time due to cache/memory accesses
overlapping access latency for one item with other work

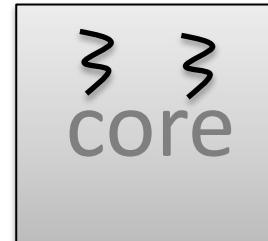
types of hardware parallelism

implicit parallelism

explicit parallelism

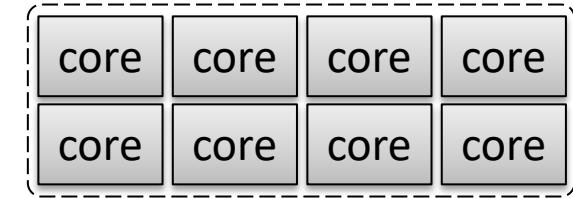


thread
what runs
your programs



instruction & data parallelism
hardware does this automatically

multithreading
threads share
execution cycles
on the same core



multicores
multiple threads run in
parallel on different cores

implicit parallelism → (almost) free lunch

explicit parallelism → must work hard to exploit it

summary – hardware parallelism

Hardware gives different parallelism opportunities.

Database systems used to be ignorant of this parallelism because it used to be implicit.

Today, data management systems do not have this luxury because we also have explicit parallelism.

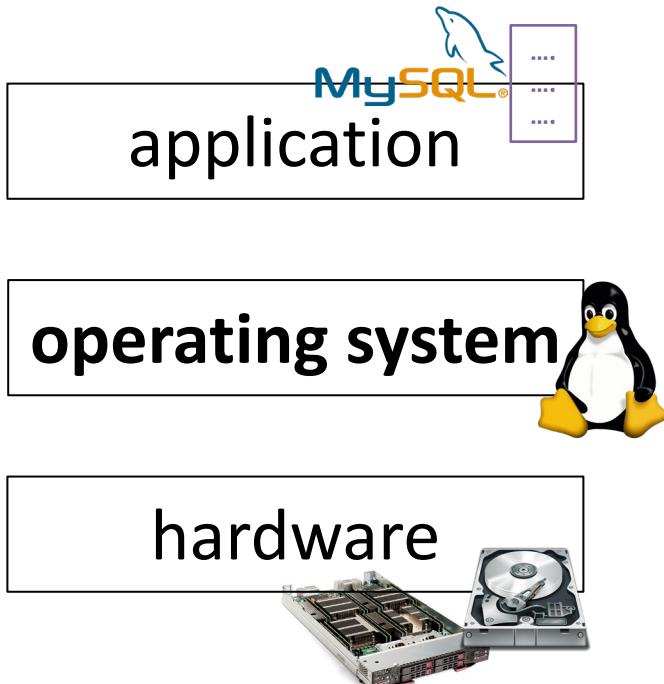
Explicit parallelism also complicates memory hierarchy.

goal: design systems that are aware of the hardware parallelism (ideally all types of it) & its implications!

agenda

- storage hierarchy
- hardware parallelism on multicores
- **operating systems**

why do we need operating systems?



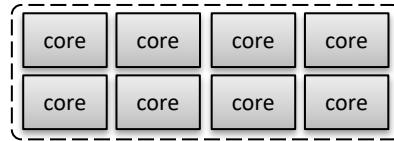
resource management

- many applications running
- many users want to use them
- hardware resources are limited

need something to reliably & efficiently share hardware resources across many users

what are the resources?

CPU



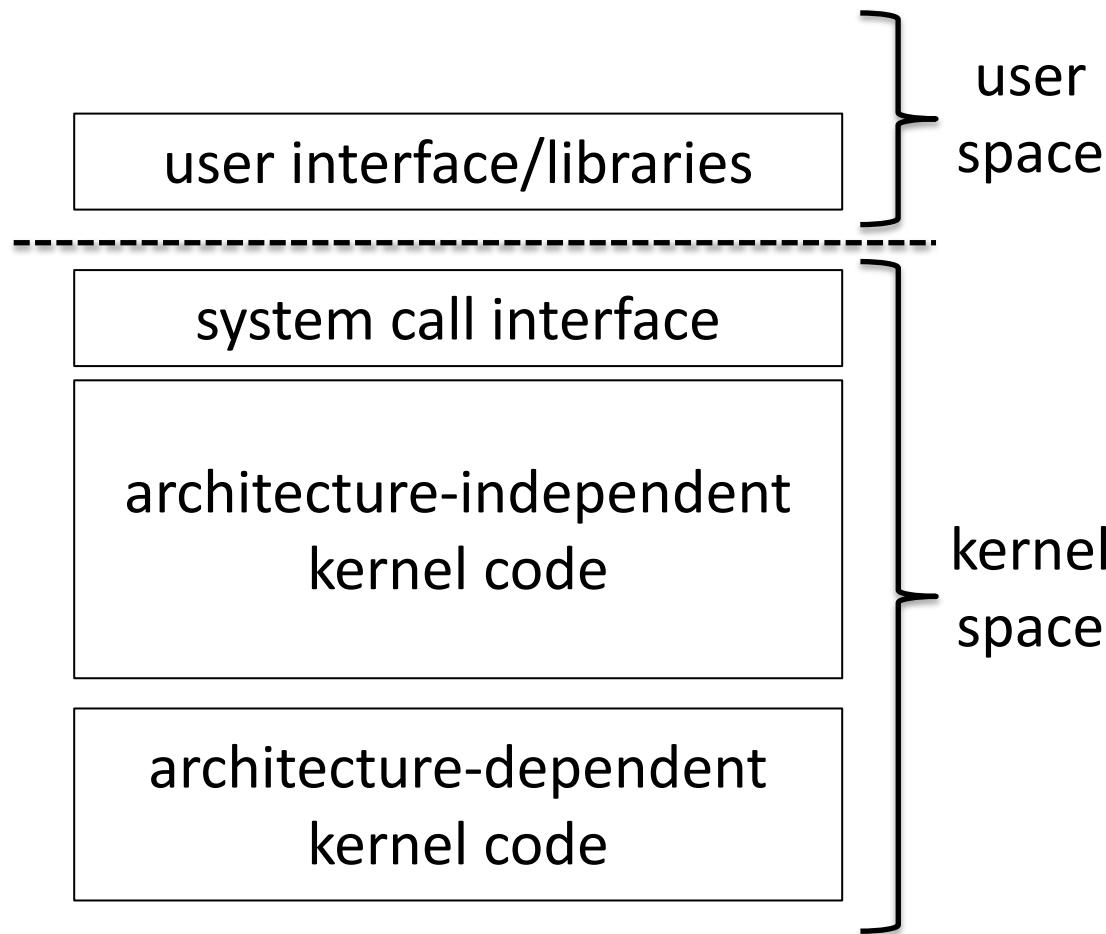
memory



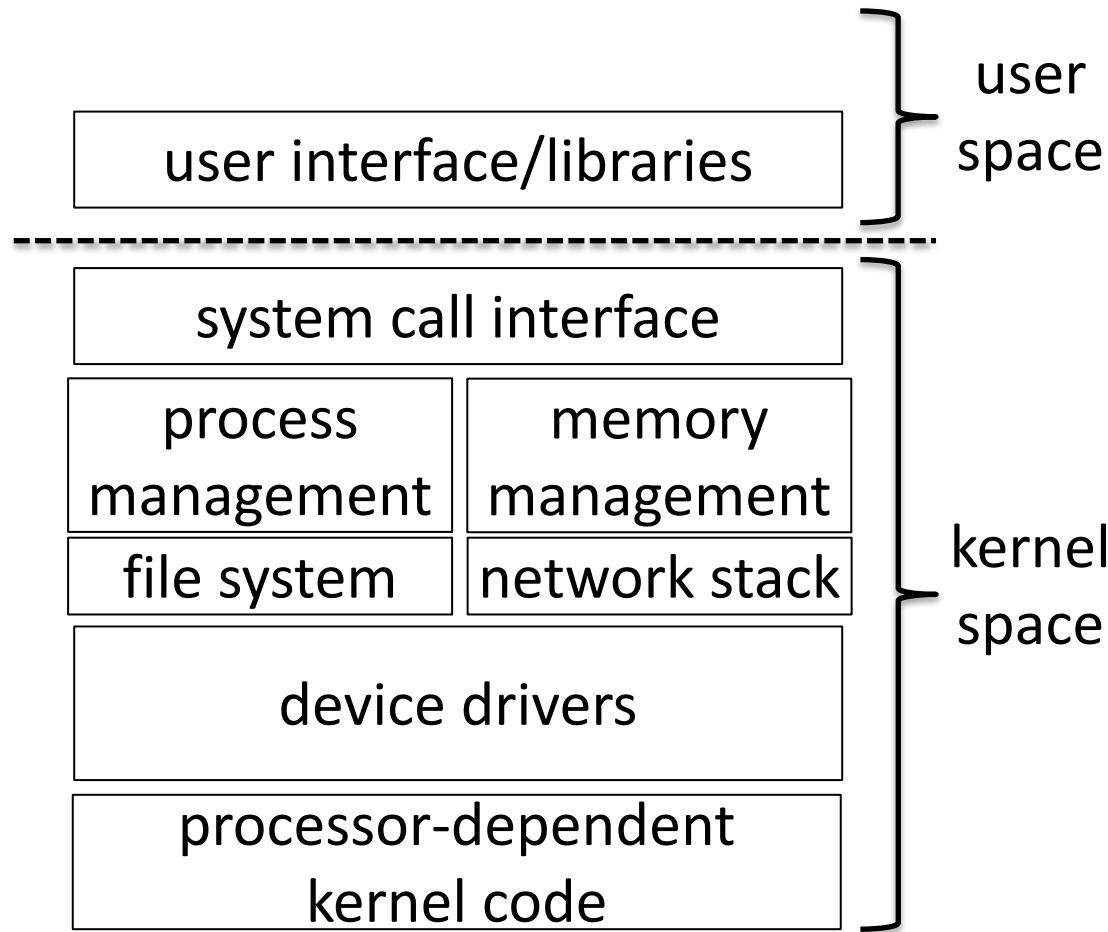
I/O devices



typical operating system components



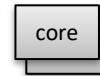
typical operating system components



process management

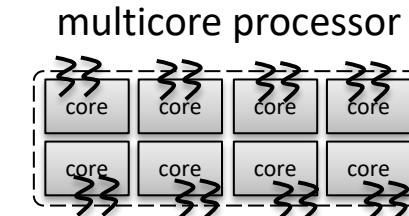
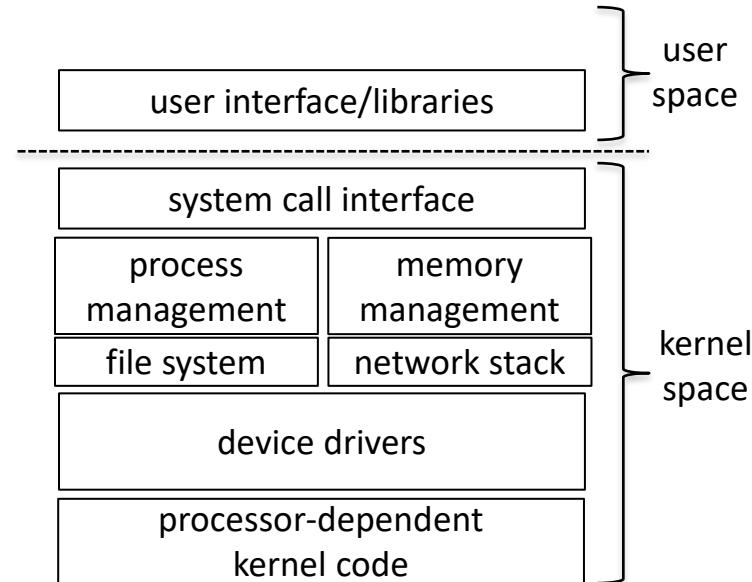
virtualizes a processor
giving the illusion of infinite #cores

user space has application(s)
an application has process(es)
a process has *thread(s)* ↗
threads are mapped to *cores*



by default operating systems handles this mapping
application can request to run on specific cores

```
int sched_setaffinity(pid_t pid, // thread id
                      size_t cpusetsize, // sizeof(cpu_set_t)
                      const cpu_set_t *mask); // bitmap indicating cores you want the thread to run on
or taskset, numactl command line options
```



process management

#threads that can be active at a time

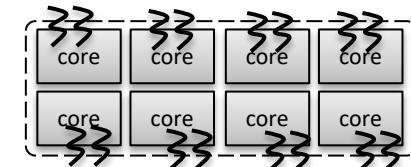
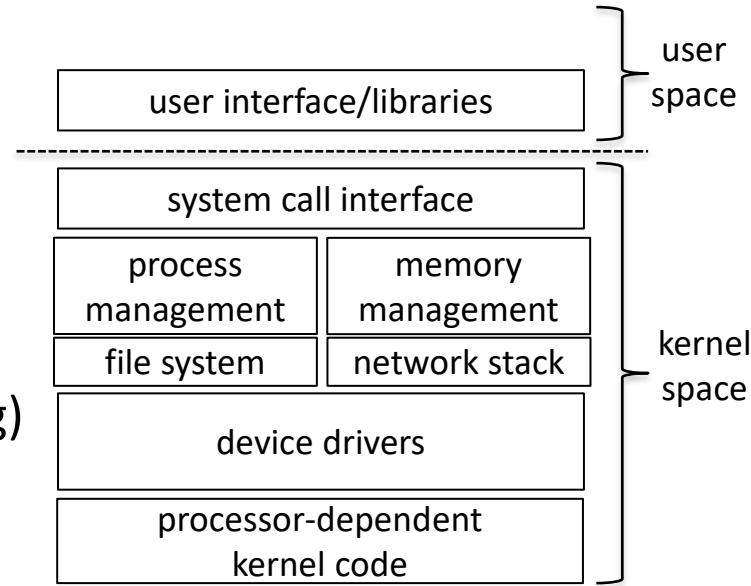
→ #hardware contexts (*logical cores*)

a processor supports

- typically 2 per core on Intel (hyper-threading)
- Sun Spark T2 had 8 per core

what if we have more threads to run?

- operating system **context switches** to swap threads' state/context in & out
if one thread is inactive (due to IO or sleeping), another one runs
- works well for general-purpose scenarios
you typically have IOs that interrupt a thread



memory management

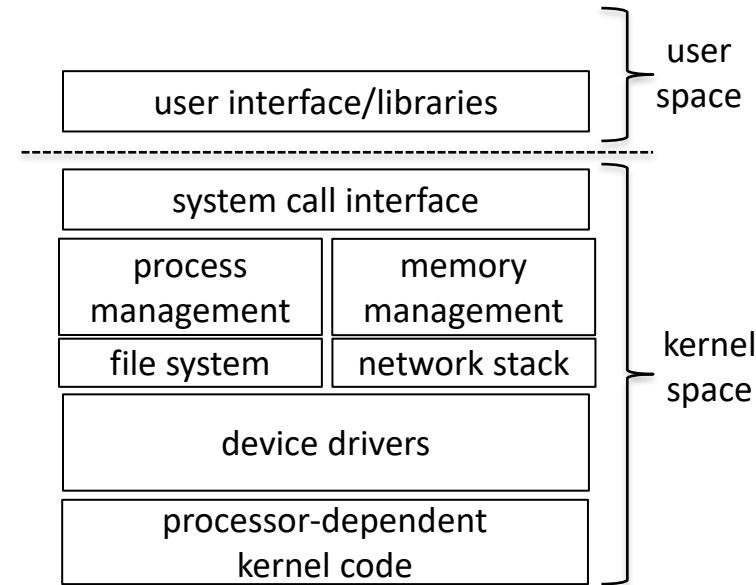
virtualizes memory
giving the illusion of infinite memory

user space has application(s)
an application has process(es)
a process has its own

memory address space = virtual memory

by default operating systems maps a process' address space to the available bytes of physical memory

- manages free space, segmentation ...
- numactl command line tool also allows binding a process to a memory region



physical memory = array of bytes

memory management

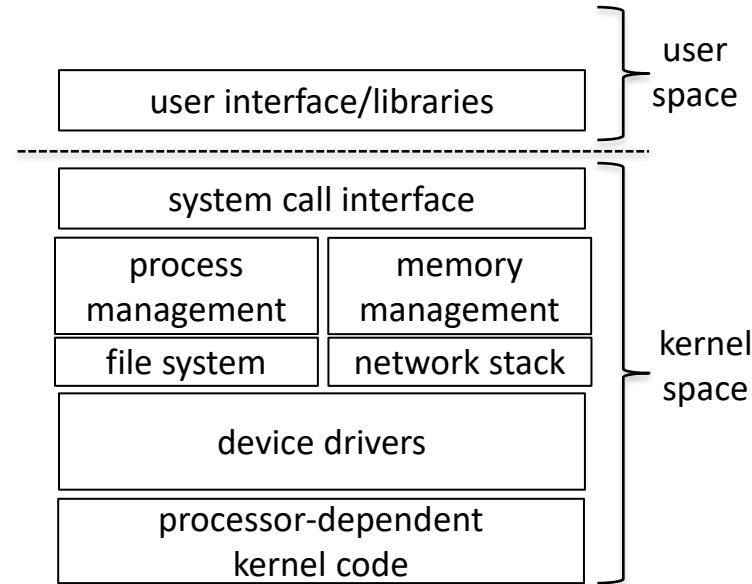
virtualizes memory
giving the illusion of infinite memory

user space has application(s)
an application has process(es)
a process has its own

memory address space = virtual memory

aggregate memory used by all processes can be larger than
available physical memory

- operating system swaps things back & forth as needed
→ to/from swap space on disk
- if has to be done too frequently, your program won't perform well



memory management

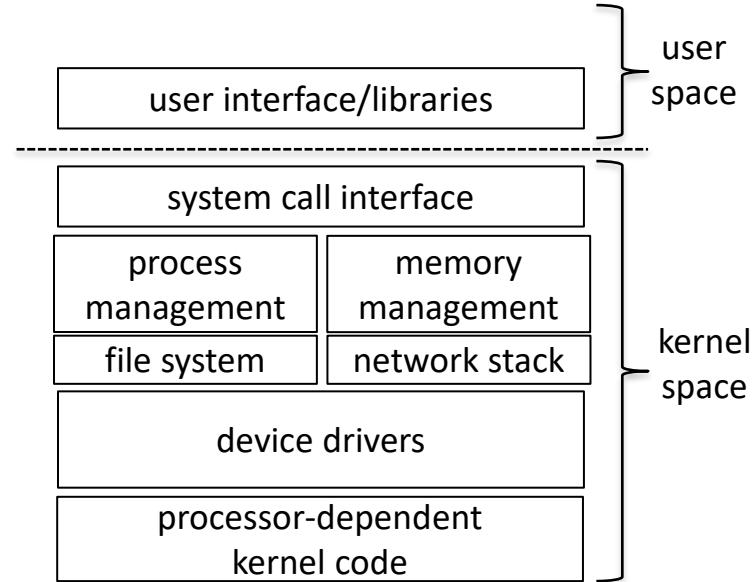
virtualizes memory
giving the illusion of infinite memory

user space has application(s)
an application has process(es)
a process has its own

memory address space = virtual memory

threads in a process share the same memory space
it is application's responsibility to manage this shared space reliably

- use locks/mutexes/atomics
- partition this space to each thread



summary – operating systems

Operating system is a **resource manager virtualizing hardware resources** for applications/end-users.

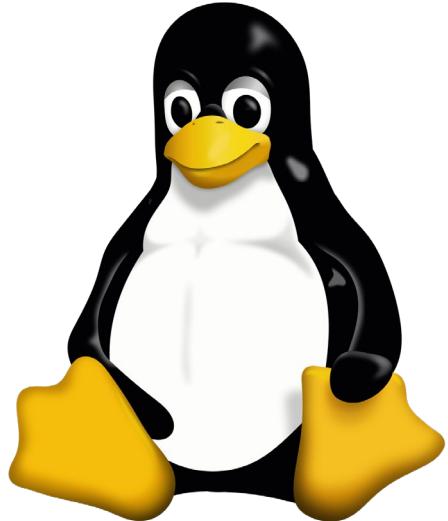
More specifically, its goal is to manage hardware resources reliably & efficiently for many applications/end-users who are using these resources concurrently.

Operating system also provides an **abstraction** layer for applications to have a common and **easy-to-use interface** while interacting with a variety hardware resources / devices.

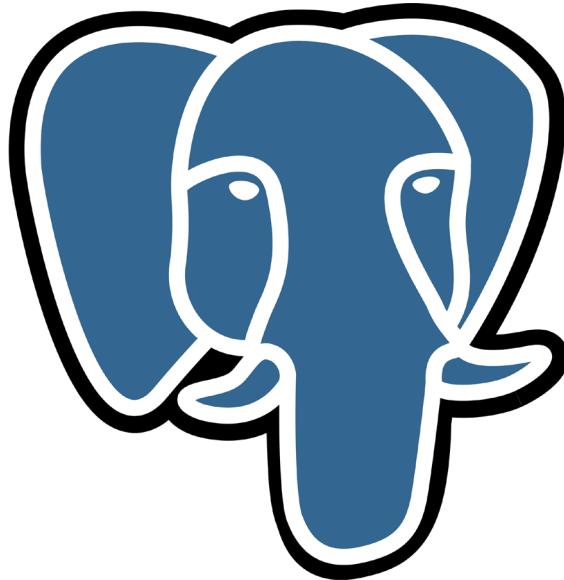
Virtualizations & abstractions come at a cost, though!

- indirect management of hardware resources
- need to think about trade-offs of the indirection

OS vs. DB



vs.



backup

what does sequential mean on hard disk?

(1) reach the first desired block

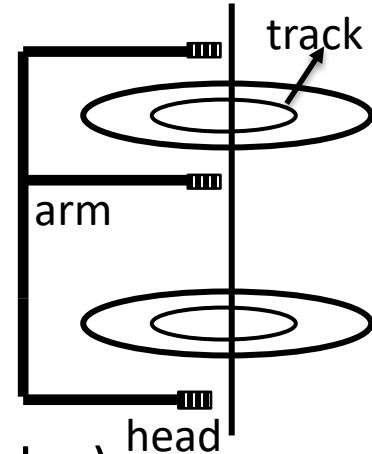
(2) read adjacent blocks on the same track

(3) read blocks on the same cylinder

(switch to different disk head, then short rotational delay)

(4) read blocks on the adjacent cylinder

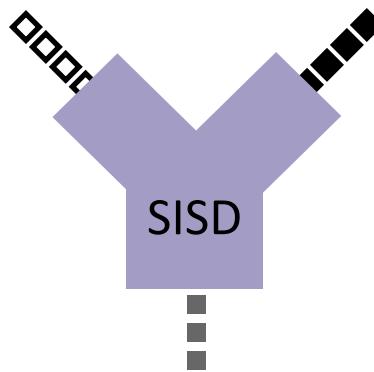
(short-distance seek time, then short rotational delay)



single instruction multiple data (SIMD)

instructions

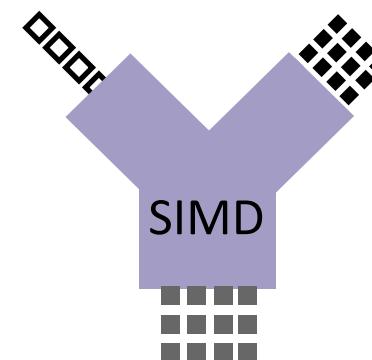
data



results

instructions

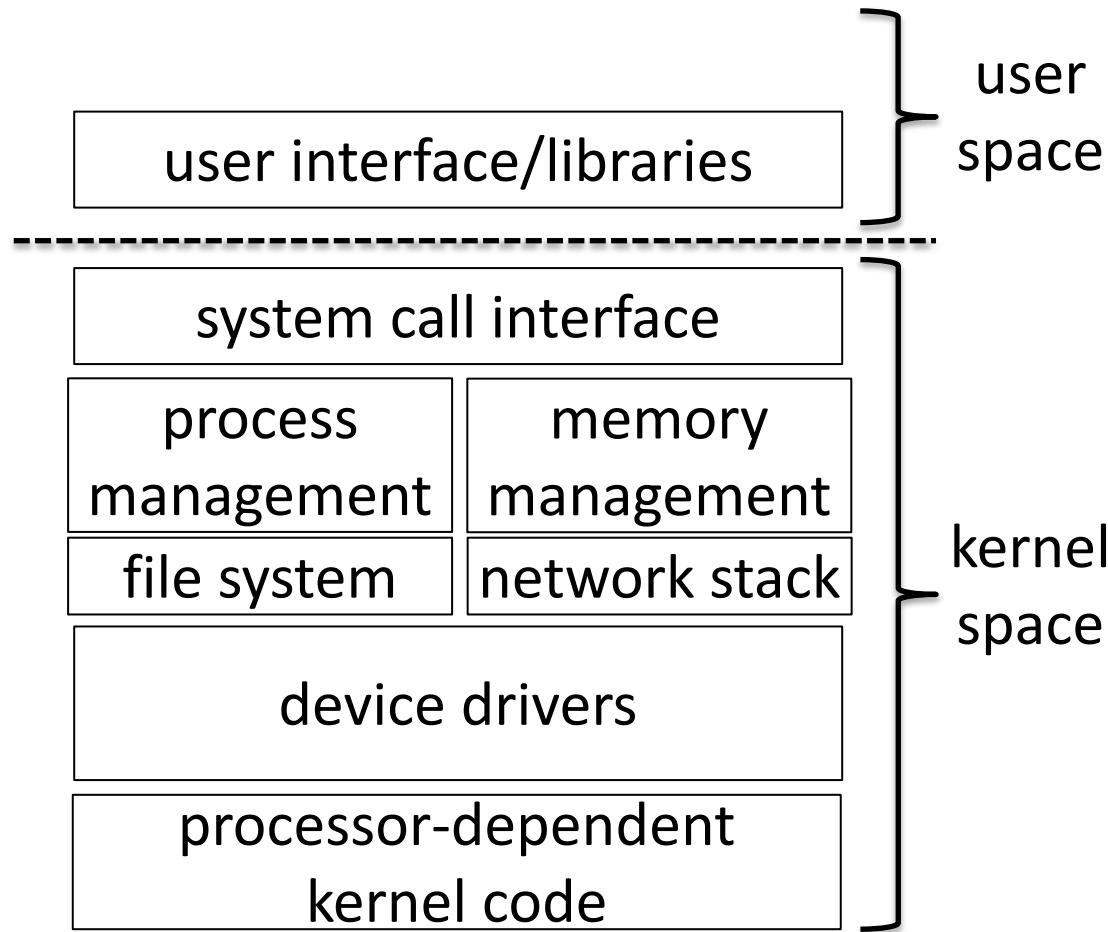
data



results

**GPUs are like SIMD machines
they support extreme parallelism**

typical operating system components

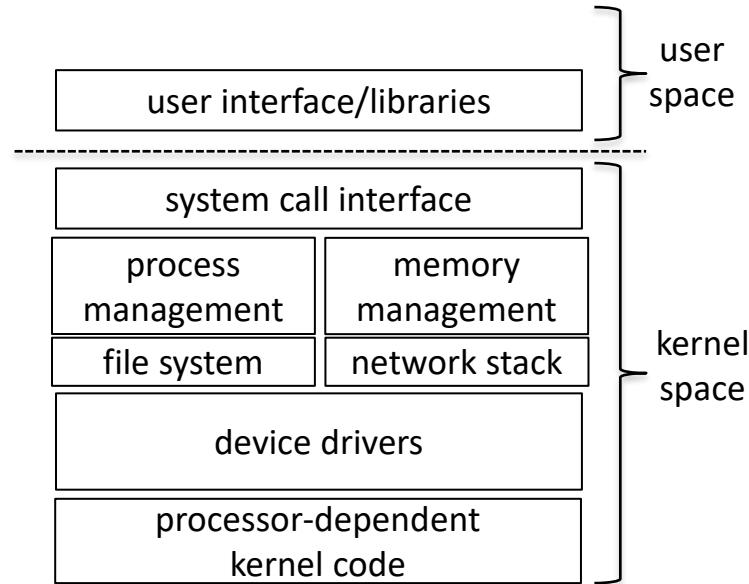


user interface/libraries

gives applications uniform & easy-to-use primitives to communicate with the kernel

thanks to these we don't have to express ourselves in assembly

- compilers (e.g., gcc)
- shells (e.g., bash)
- GNU libraries in general

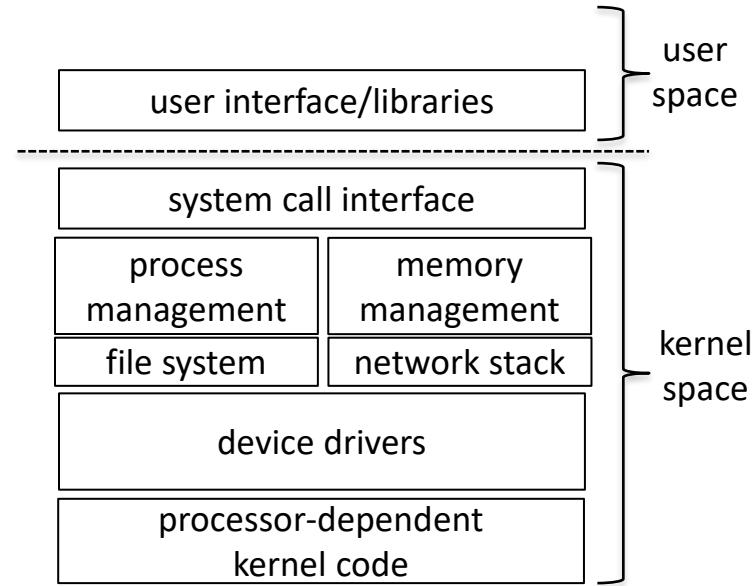


system call interface

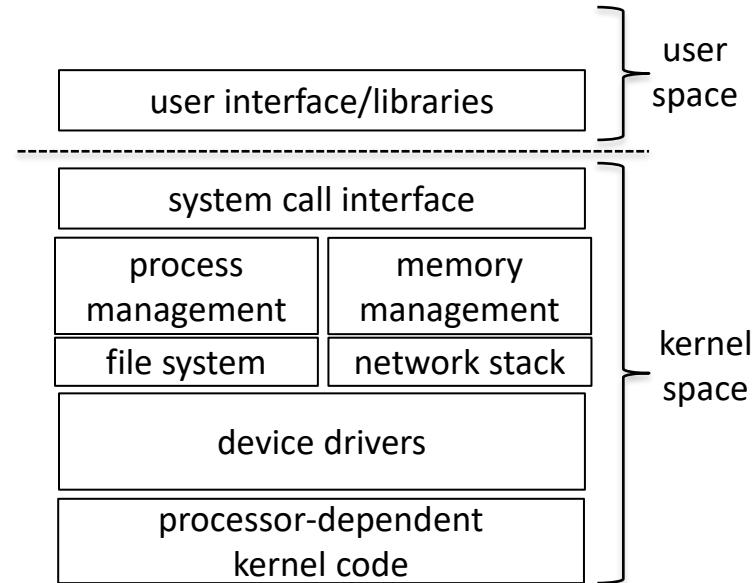
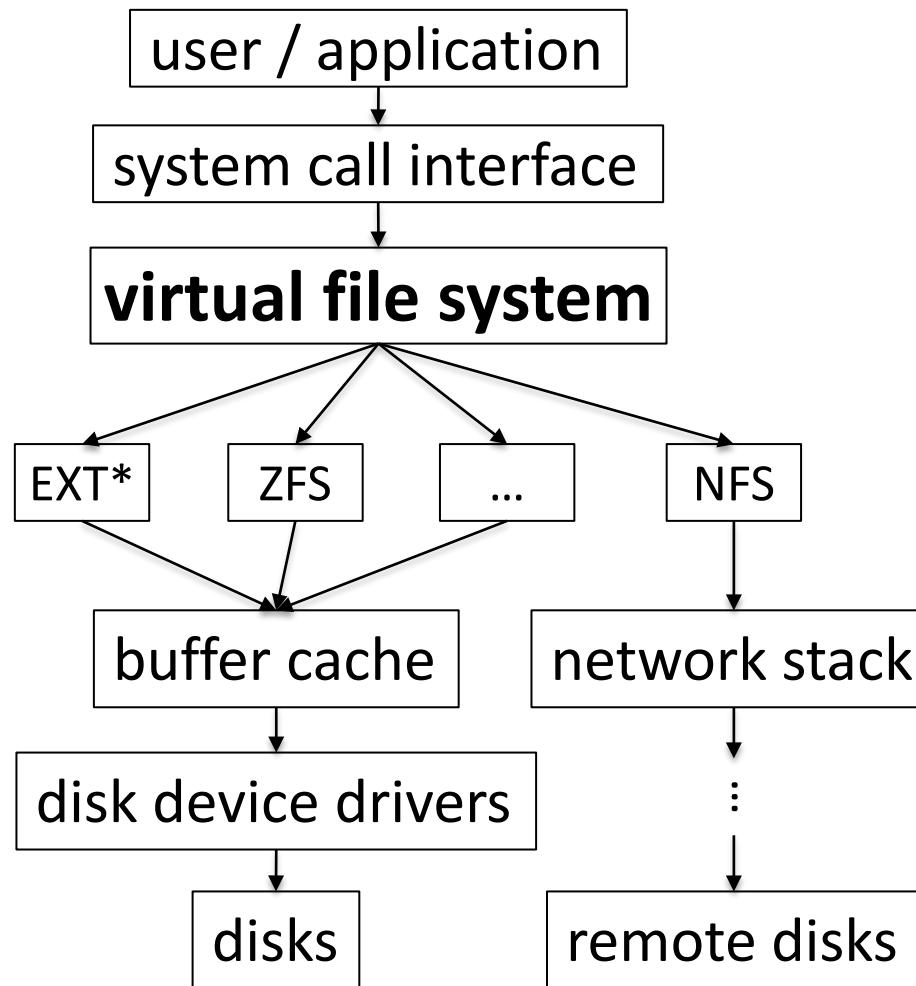
helps separating user space
from kernel space

kernel talks to hardware since it
manages hardware resources
for many user requests

users just make request such as
reading/writing a file,
allocating more memory,
sending packets over network,
creating a new process, etc.

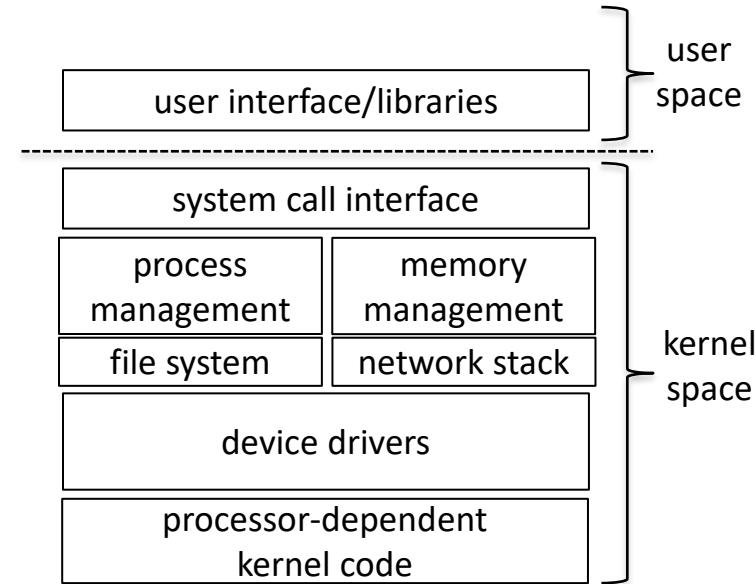
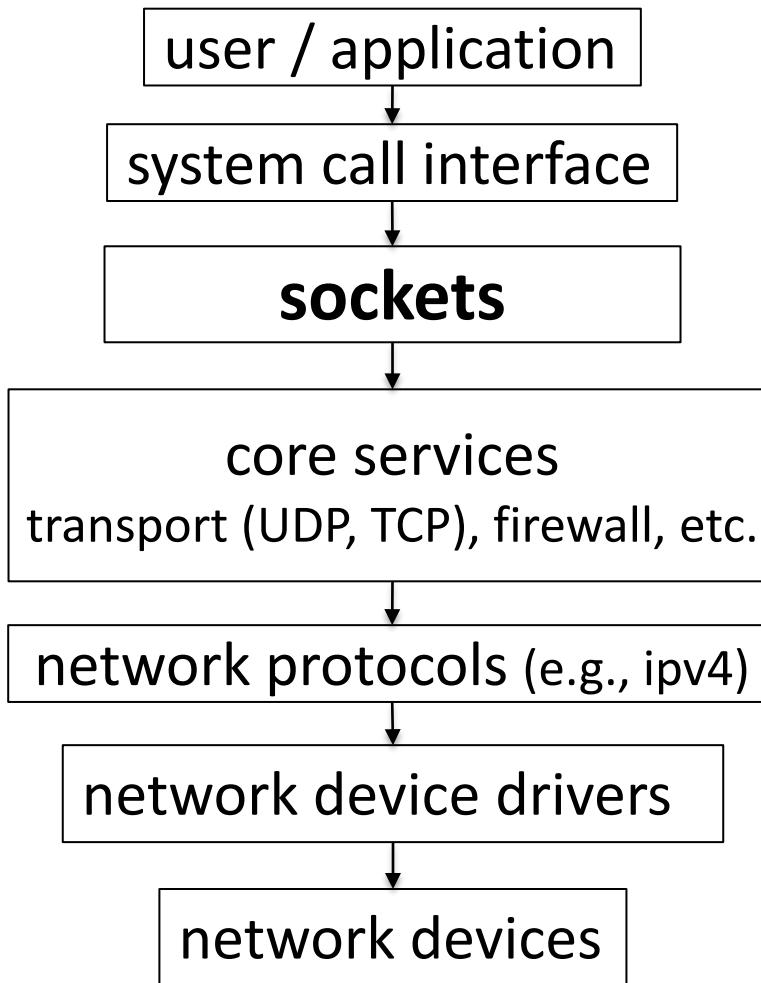


file system



enables common interface to access different file systems

network stack

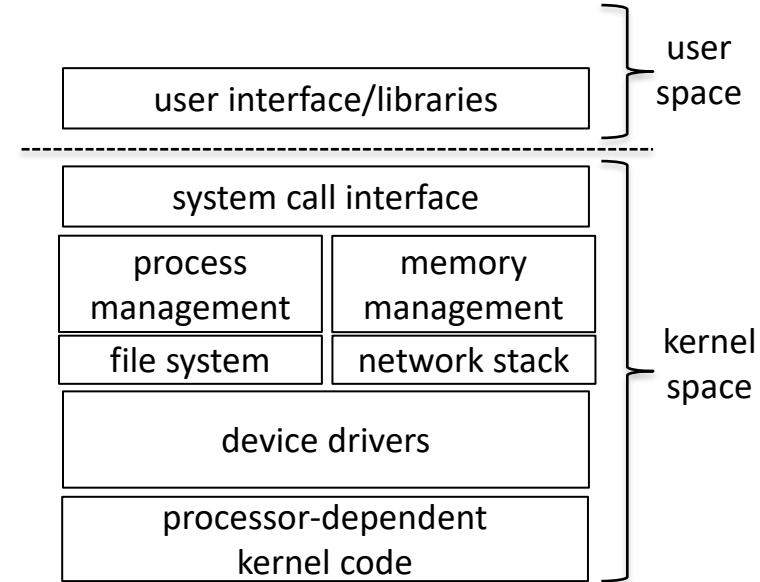


socket model enables a common interface to different protocols

device drivers

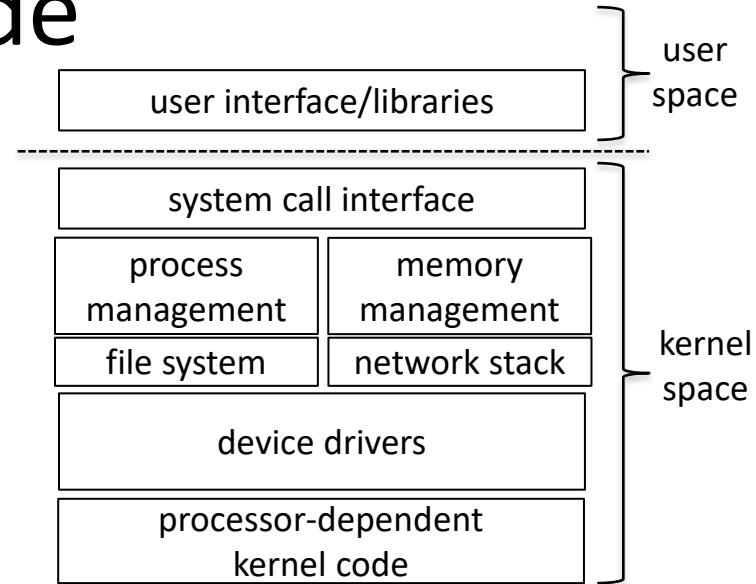
software interface to hardware devices

a bloated component of the kernel
since it is both hardware dependent
& operating system specific



processor-dependent code

- different processors support different instructions sets
- certain processors support functionalities like transactional memory, SIMD, memory alignment ...
- desktop vs. server



need to be able to handle these differences without changing majority of the kernel code

Introduction to Database Systems

I2DBS – Spring 2023

- Week 8:
- Indexes

Jorge-Arnulfo Quiané-Ruiz

Readings:

PDBM 12

Information

- **Homework 2:**
 - We will provide feedback and solution ASAP
- **Exercise 8 is online**
 - It is about impact of indexes
 - = Topic for today (and partly next week)
- **Homework 3**
 - It will be online early next week (or before)
 - Yet, you have 2 working weeks to solve it: deadline on April 21st
 - Topic = DDL + Normalization + Indexing (+ SQL)
 - Rather extensive ... but you need to learn all of this!
 - Exercise 8 is very useful as preparation!



Profile of the Week

Edward M. McCreight

Co-Inventor of B-Tree Data Structure

- 1940: USA
- 1968: co-invented B-Tree
"A Space-Economical Trie Storage Structure"
- 1969: PhD in Computer Science from Carnegie Mellon
- **Institutions:** Boeing, Xerox, Adobe





*Maintain
& tune*



RDBMS

Database Administrator
(DBA)

Indexes

Readings:

PDBM 12

Initial Case Study

- **Which of these queries should run faster?
How much faster?**

(1) SELECT COUNT(*) FROM movie
WHERE year=1948;

(2) SELECT COUNT(*) FROM movie
WHERE year=1920 or year=1924 or year=1928 or year=1932
or year=1936 or year=1940 or year=1944 or year=1948;

(Q1) If ran several times, would the first run be slower? Why?

(Q2) If so, how big would be that difference?

(Q3) Would an index improve performance?

Seeing Whether an Index is Used

- EXPLAIN ANALYZE can be used to show PostgreSQL's query plan

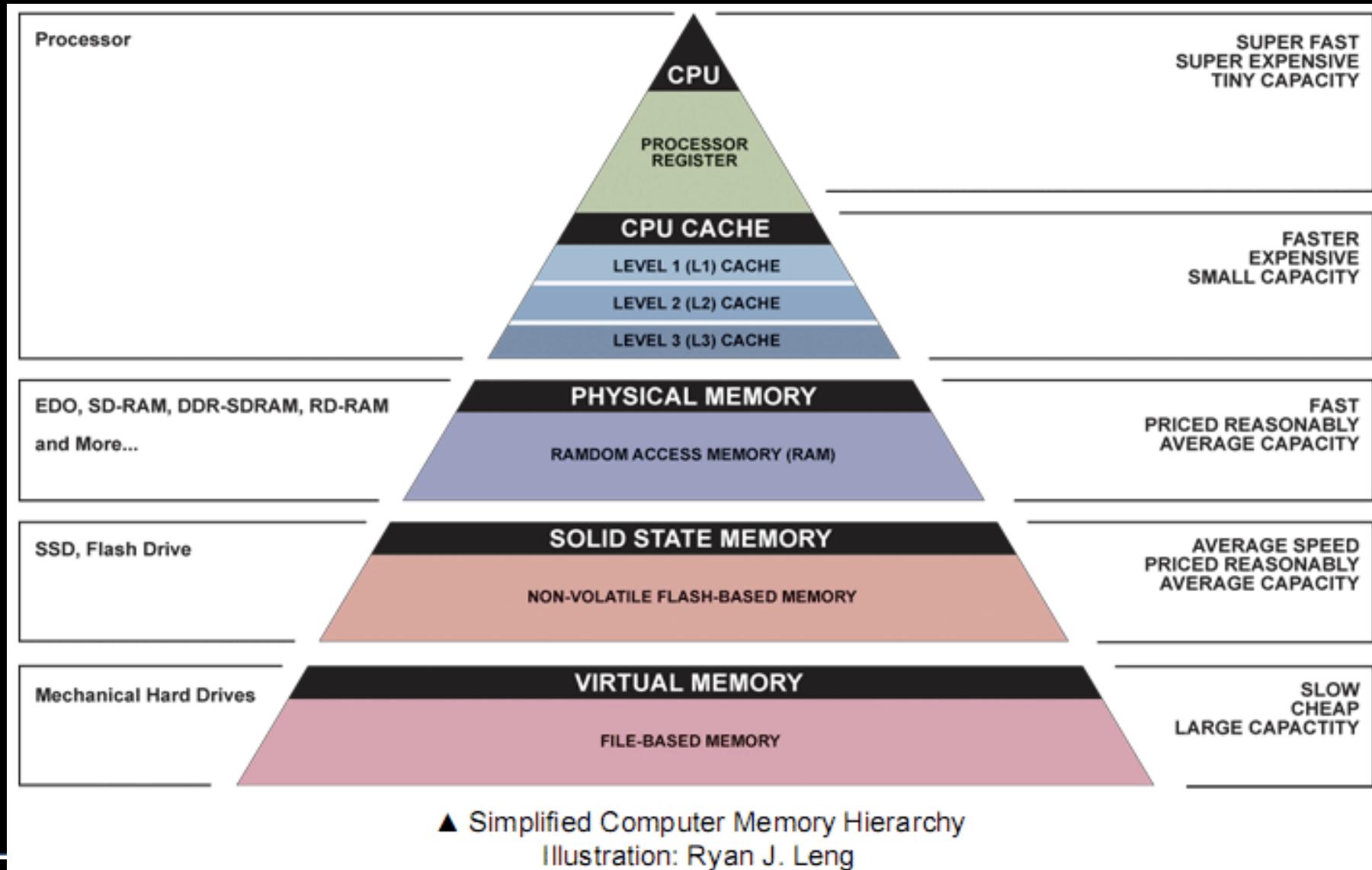
(1) EXPLAIN ANALYZE

```
SELECT COUNT(*) FROM movie  
WHERE year=1948;
```

(2) EXPLAIN ANALYZE

```
SELECT COUNT(*) FROM movie  
WHERE year=1920 or year=1924 or year=1928 or year=1932  
or year=1936 or year=1940 or year=1944 or year=1948;
```

Disk Recap



Impact of Disk-Based Storage

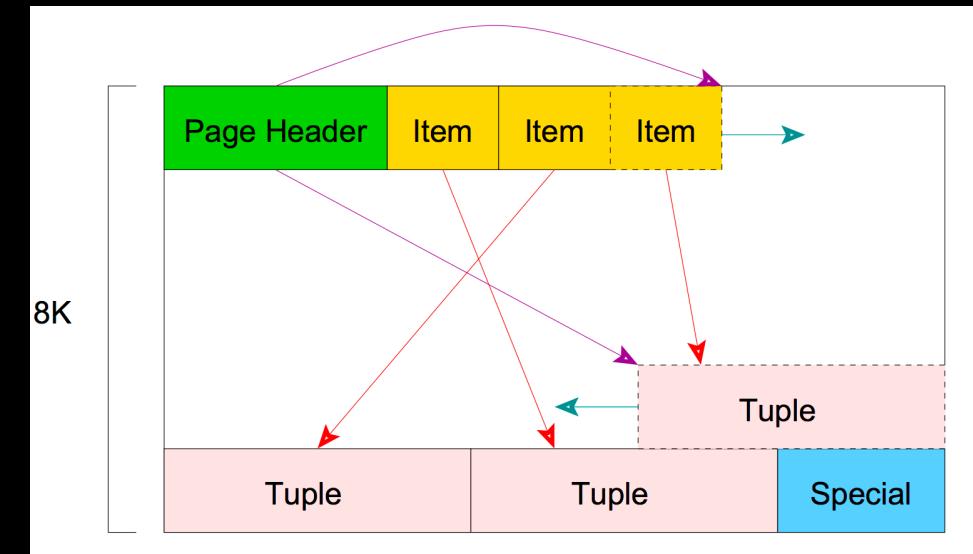
- **Unit of disk reads: KBs**

- We cannot read a few bytes from disk!
- Traditional DBMS: 8KB / 16KB
- Linux: 128KB
- How many employee records fit in one disk read?

- **Historically:**

Avoid reading in random order

- Penalty 1: Random reads costly (but SSDs!)
- Penalty 2: Have to read the same page often!



Random or Sequential?

- **Example:**

- Table = 2B rows = 1 TB
- Memory = 256 GB ⇒ only small part of the table fits in RAM!
- Disk read = 128 KB ⇒ 8M sequential reads vs 2B random reads!
- Cost of reading records in sequential or random order?

- **Costs:**

- HDD:
 - Sequential = ~8M SRs = 1.5 hours
 - Random = ~2B RRs = 114 days
- SSD:
 - Sequential = ~8M SRs = 7 minutes
 - Random = ~2B RRs = 30 hours

	ms/IO	I/Os	msec	sec	min	hours	days
HDD	0,642	8.000.000	5136000	5136	86	1,43	0,06
	4,930	2.000.000.000	9860000000	9860000	164.333	2.738,89	114,12
SSD	0,056	8.000.000	448000	448	7	0,12	0,01
	0,055	2.000.000.000	1100000000	110000	1.833	30,56	1,27

Full Table Scans

- When a DBMS sees a query of the form:

```
SELECT *  
FROM R  
WHERE <condition>
```

It reads through all the tuples of R and report those tuples that satisfy the condition.

Selective Queries

- Consider the query from before:

```
SELECT *  
FROM R  
WHERE <condition>
```

- If we have to report 80% of the tuples in R, it makes sense to do a full table scan.
- On the other hand, if the query is very selective, and returns just a small percentage of the tuples, we might hope to do better.

Point Queries

- Consider a selection query with a single equality in the condition:

```
SELECT *  
FROM person  
WHERE birthdate='1975-02-06';
```

- This is a **point query**: We look for a single value of birthdate.
 - We may still return > 1 record!
- Point queries are easy if data is sorted by the attribute used in the condition.
 - How? What algorithms would work?

Range Queries

- Consider a selection query of the form:

```
SELECT *  
FROM person  
WHERE birthdate BETWEEN '1975-02-01' and '1975-02-28';
```

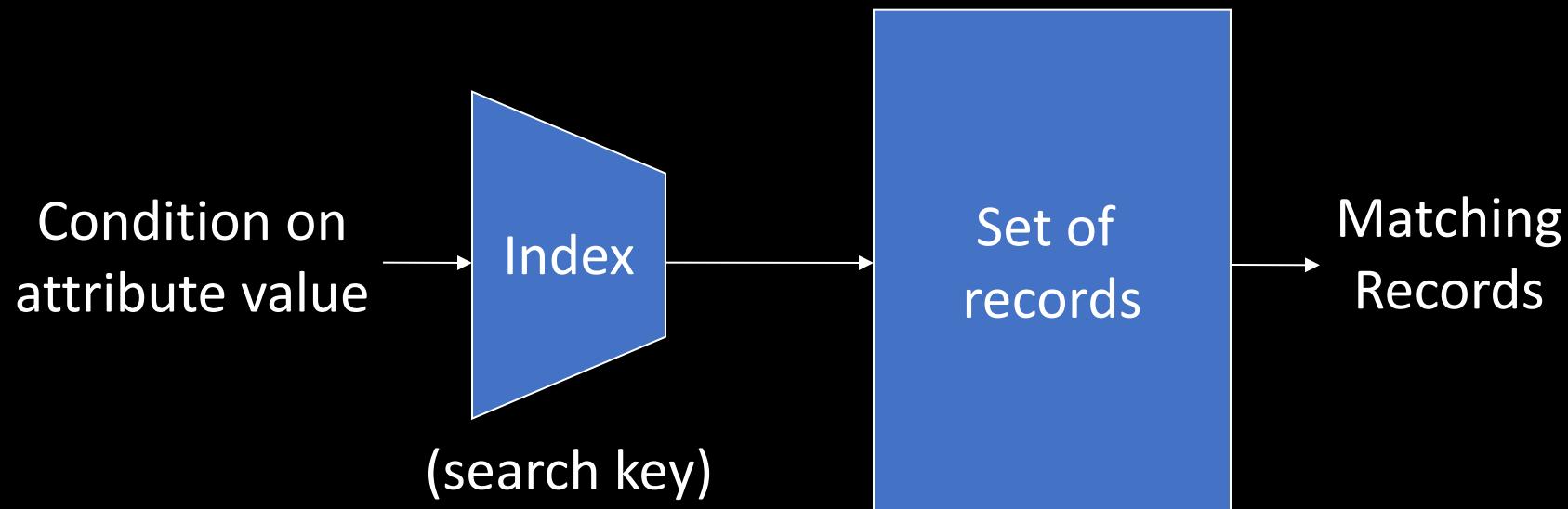
- This is a **range query**: we look for a range of values of birthdate.
- Range queries are also easy if data is sorted by the right attribute.
 - But often not as selective as point queries.

Indexes

- To speed up queries the DBMS may build an index on the birthdate attribute.
- A database **index** is similar to an index in the back of a book:
 - For every piece of data you might be interested in (e.g., the attribute value 1975-02-06), the index says where to find the row with the actual data!
 - The index itself is organized such that one can quickly do the lookup.
- Looking for information in a relation with the help of an index is called an **index scan** (range) or **index lookup** (point)

Indexing

- An index is a data structure that supports efficient access to data
 - In databases, indexes are also stored on disk



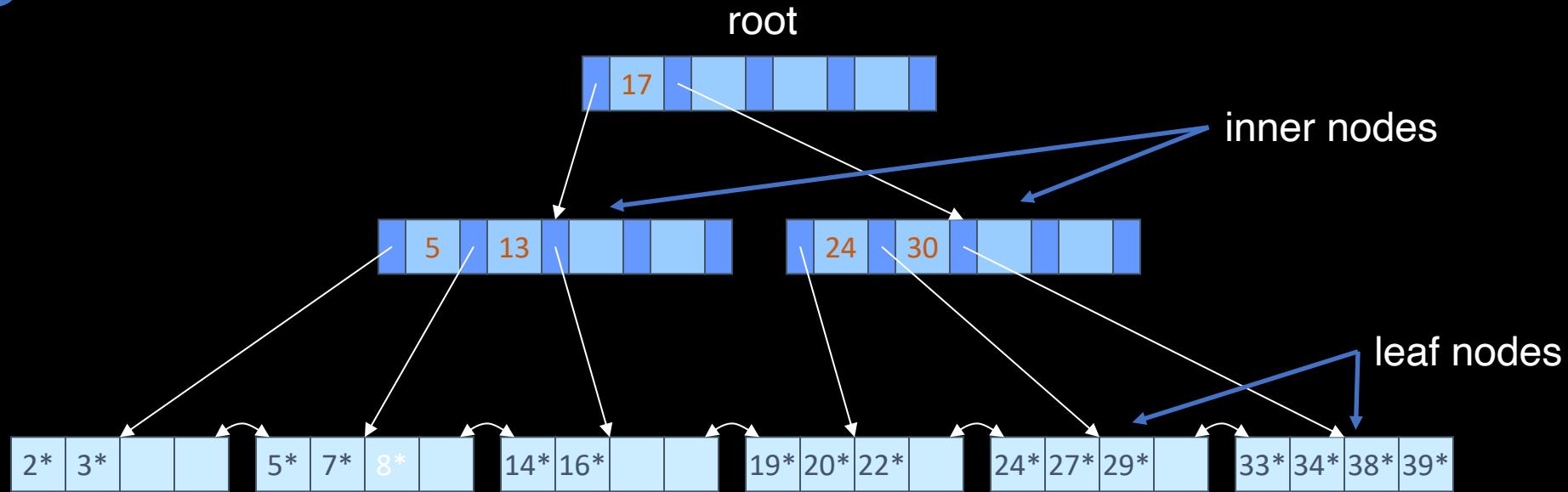
Two Techniques

- **Two basic techniques dominate in modern DBMSs:**
 - **Hashing**: Use a fixed transformation algorithm to convert the attribute value into a database address
 - **Tree search**: A dynamic search structure is built that guides the search for a given attribute value to the proper database location
- **Hashing supports equality queries only.**
 - Typically used dynamically for large-scale joins
 - Rarely available to developers
- **Tree search is more versatile and accessible**

$\mathcal{B}+$ Trees

- **The most common index type**
... in relational systems
- **Supports equality and range queries**
- **Dynamic structure**
 - Adapts to insertions and deletions
 - Maintains a balanced tree

A Sample B+-tree

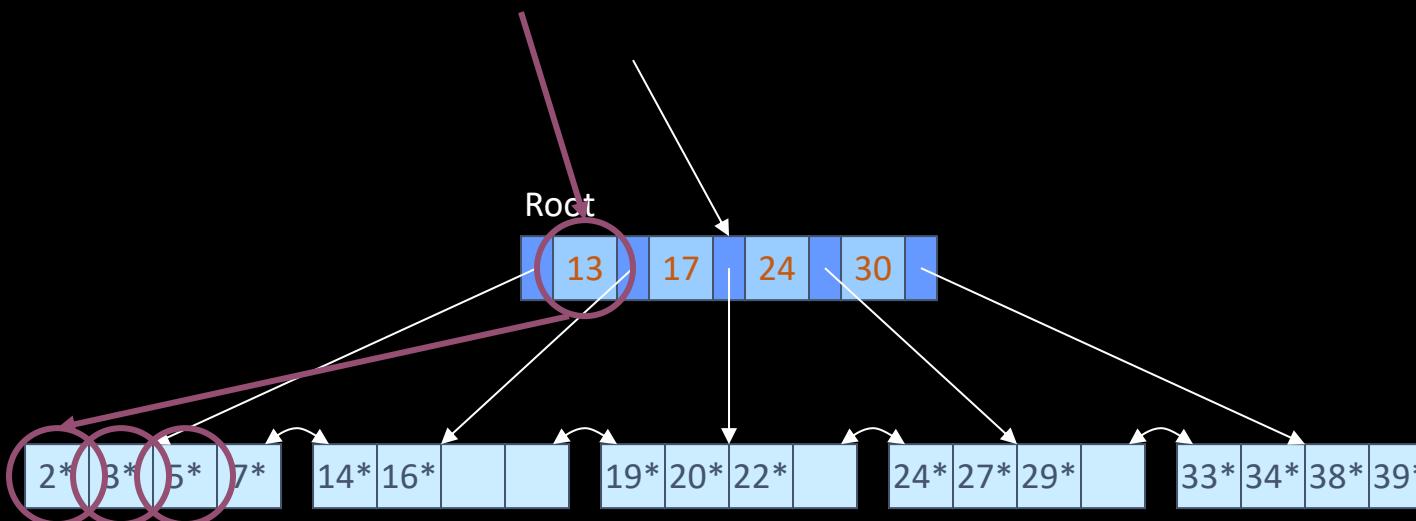


- **X* represents (search key, pointer list) pairs (X, [address of tuple X1, ...])**
 - Unique index: Only one entry in list
- **Key values are sorted: K1 ≤ ... ≤ Kd (d is maximum capacity or order of node)**
 - For any two adjacent key values Ki, Ki+1 the pointer Pi points to a node covering all values in the interval [Ki, Ki+1)

Each inner/leaf node is one disk page
Nodes have a minimum & maximum capacity

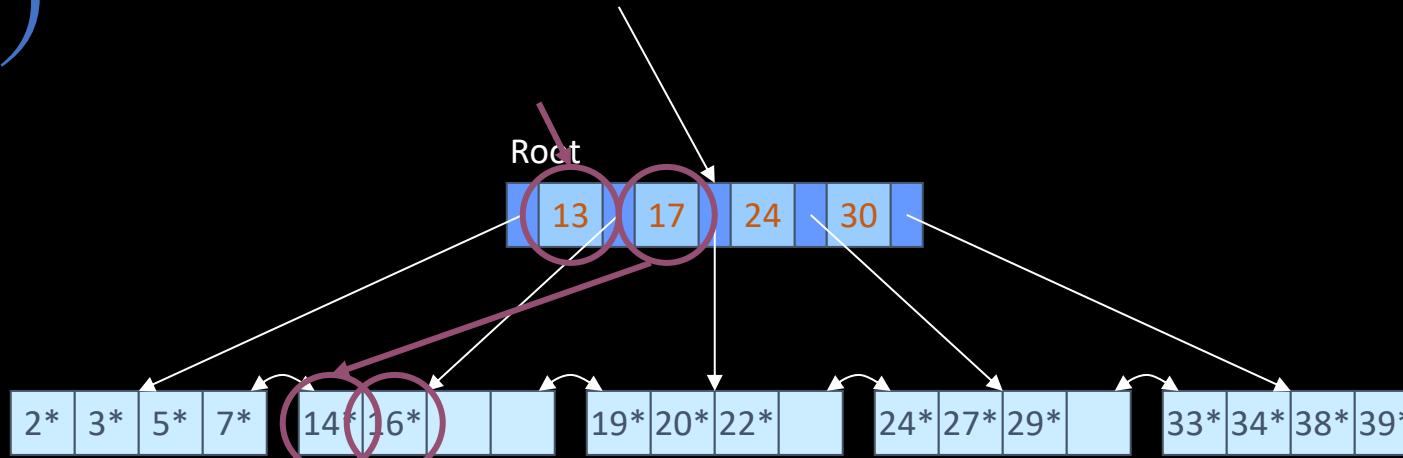
Searching

- Begin at the root
- Comparisons guide the search to the appropriate leaf
- Ex: Find 5*

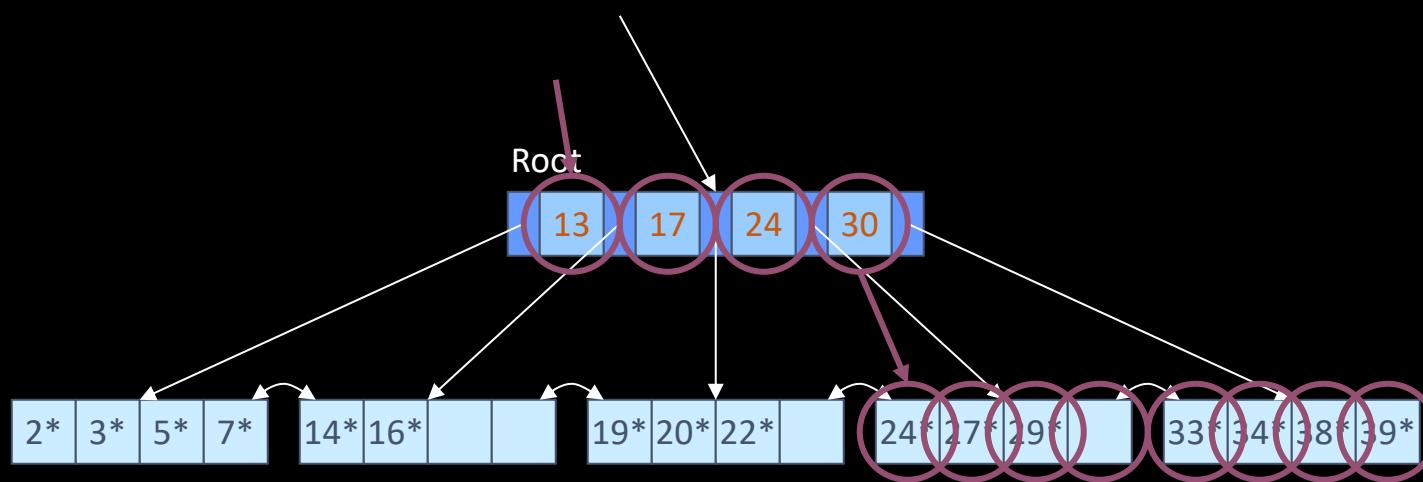


Searching (II)

- Ex: Find 15^*

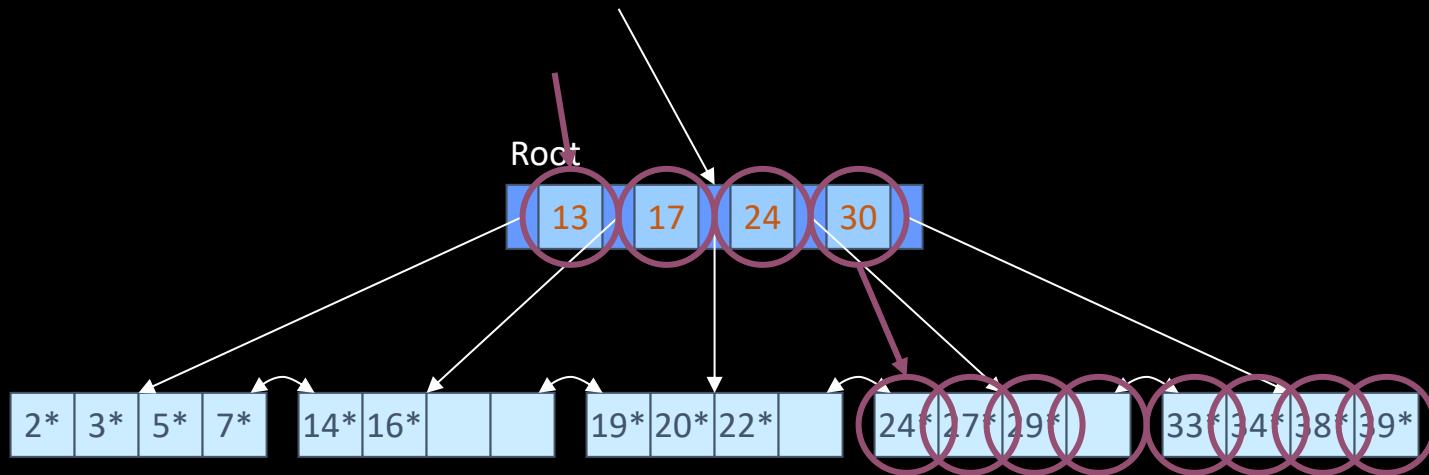


- Ex: Find all records $\geq 24^*$



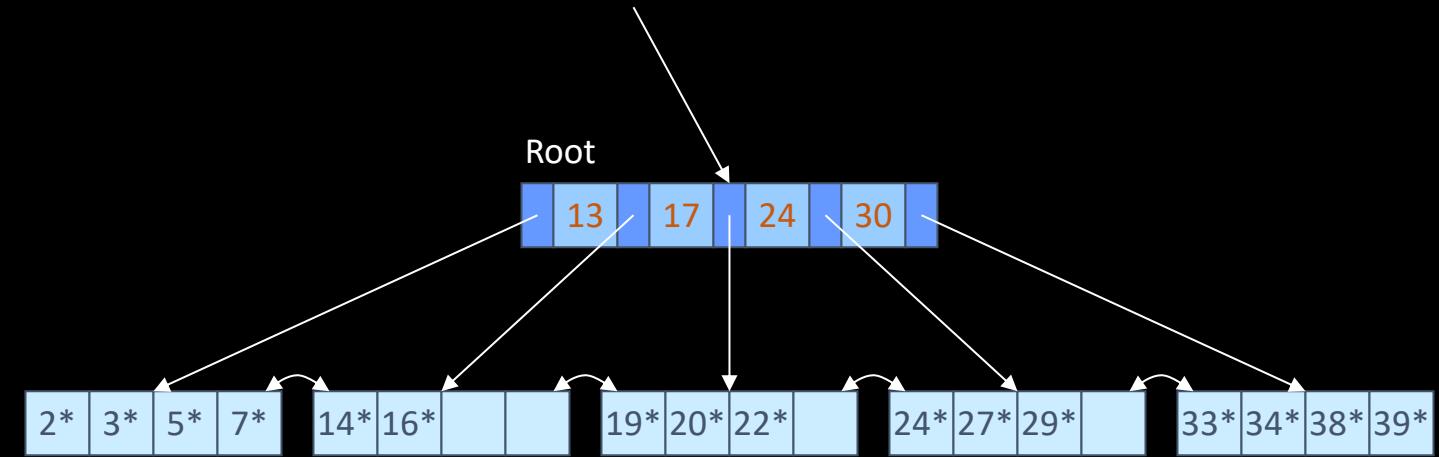
Intra-Node Searching

- We have used scans
- B+-tree nodes have hundred(s) of key values
- Use binary search!

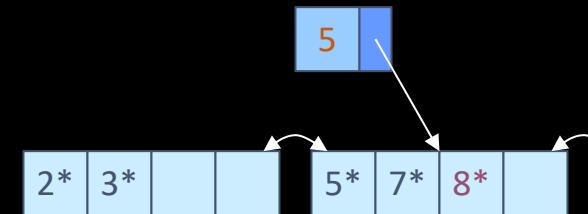


Inserting

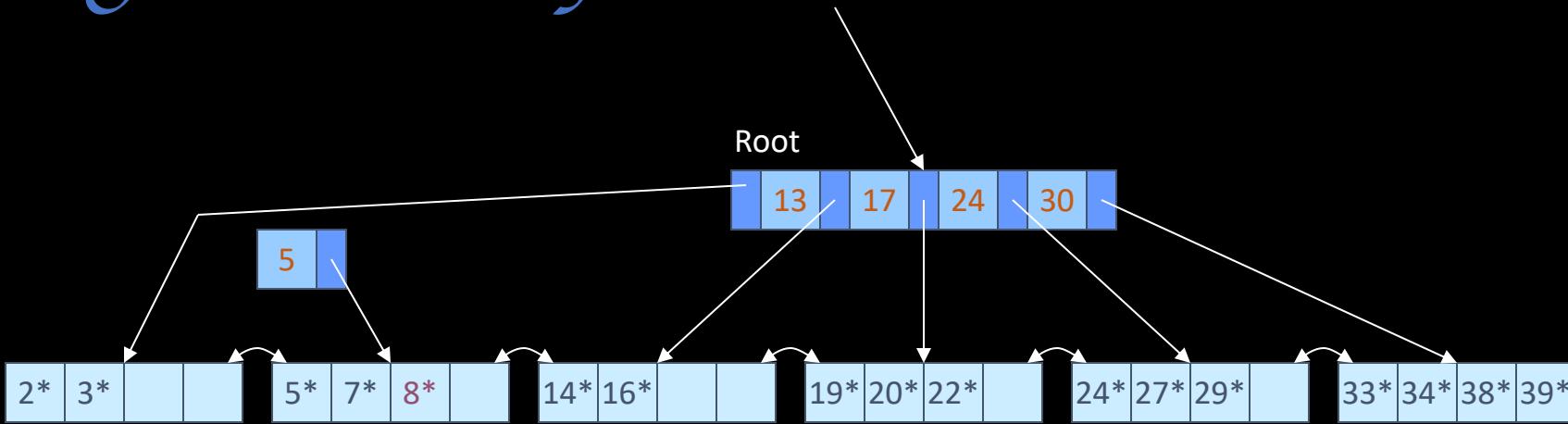
- Leaf is full, must therefore split
- Root is full, must therefore split



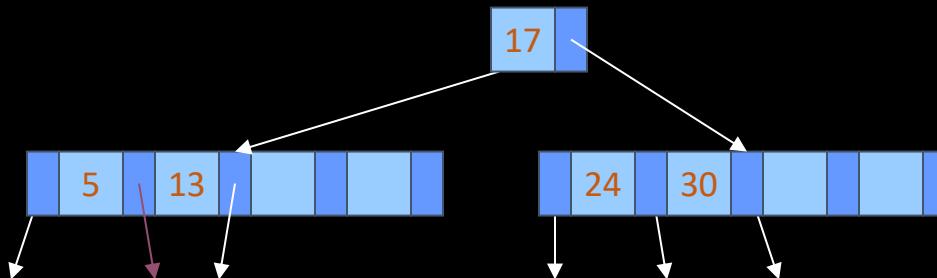
- Inserting 8* Example
 - First split the leaf
 - Copy middle search key to the parent



Inserting 8* Example (cont.)

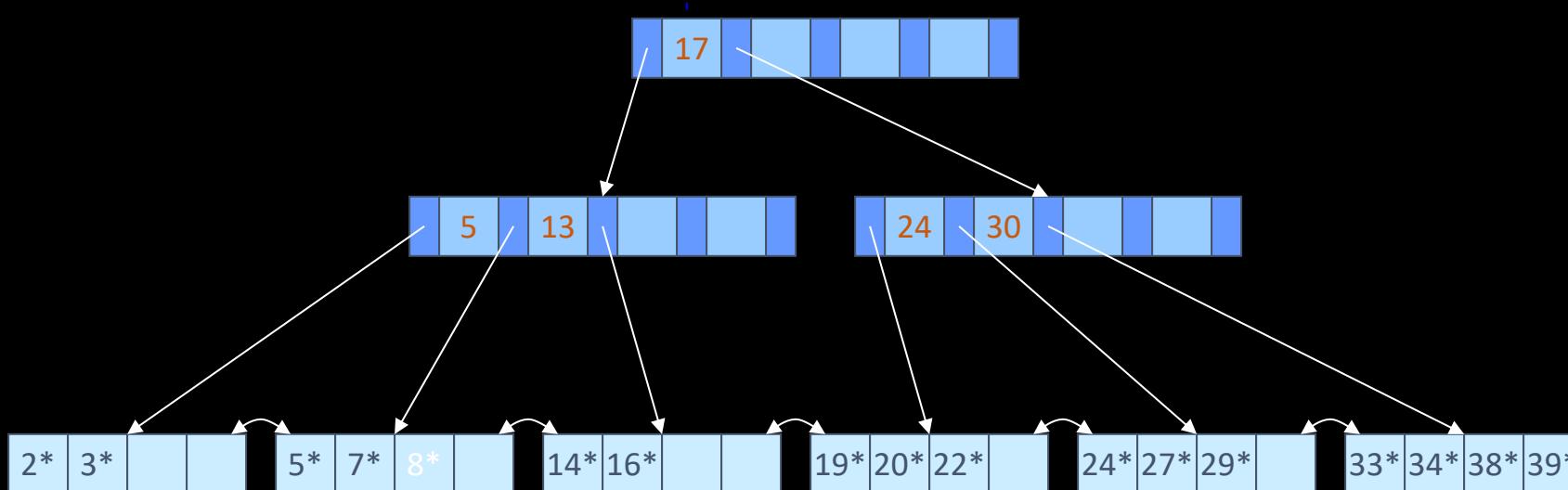


- Then split the root
- Move the middle search key into the new root



After Inserting 8*

- Trees grow wider, then higher



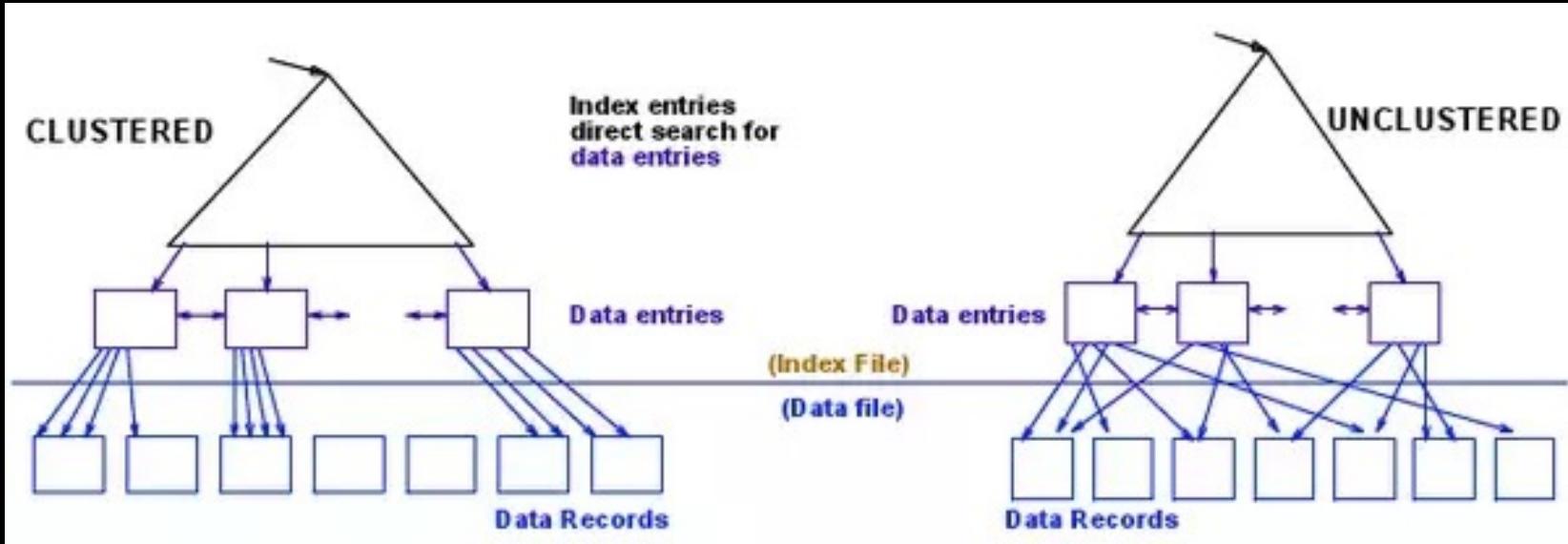
Storage Capacity - Some Statistics

- **A typical tree:**
 - Order: 1000 ($\sim= 16 \text{ KB per page} / 16 \text{ b per entry}$)
 - Utilization: 67% (usual numbers in real life)
 - Fanout: 670
- **Capacity**
 - Root: 670 records
 - Two levels: $670^2 = 448.900$ records
 - Three levels: $670^3 = 300.763.000$ records
 - Four levels: $670^4 = 201.511.210.000$ records
- **Top levels may fit in memory**
 - Level 1 = 1 page = 16 KB
 - Level 2 = 670 pages = 11 MB
 - Level 3 = 448.900 pages = 7 GB

Index Jargon

- **Indexes vs. Indices**
- **Search key vs. Primary key vs. Candidate key**
- **Unique index vs. Non-unique index**
- **Primary index vs. Secondary index**
- **Dense index vs. Sparse index**
- **Clustered index vs. Unclustered index**

Clustered vs. Unclustered Index



Clustered Indexes

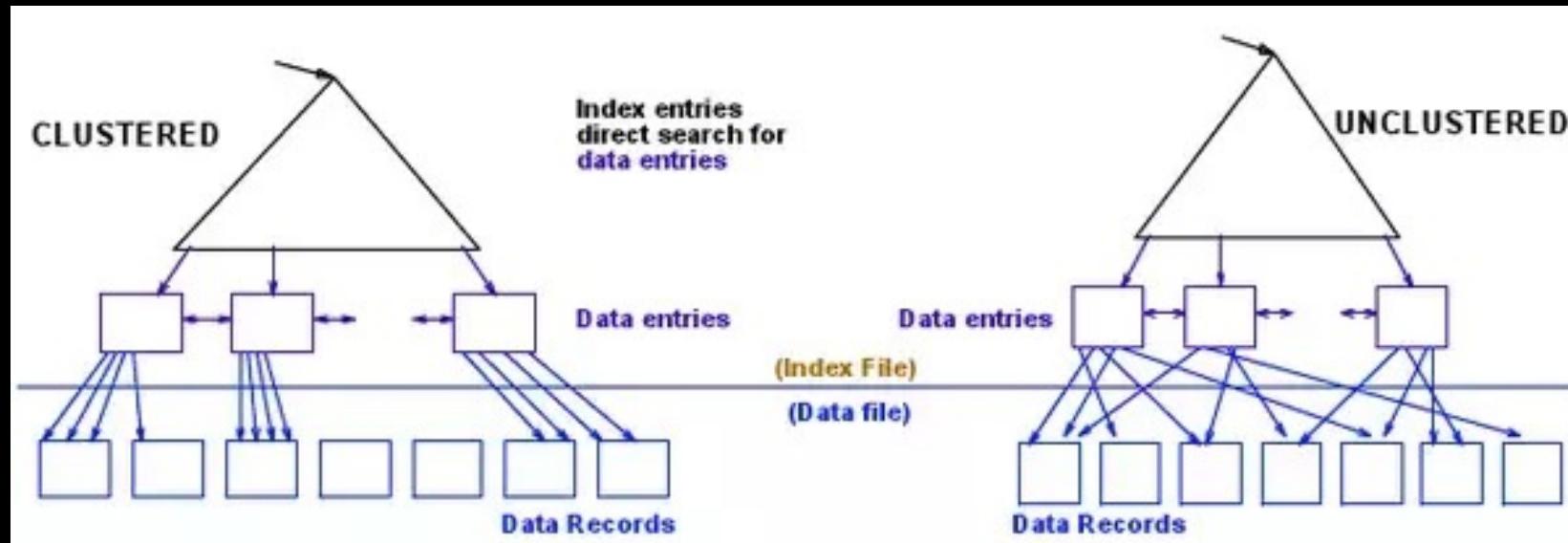
- If the tuples of a relation are stored sorted according to some attribute, an index on this attribute is called **clustered**.
 - Clustered indexes make point and range queries on the key very efficient
 - Why? Sequential reads + As few reads as possible!
- Many DBMSs automatically build a clustered index on the primary key of each relation.
 - PostgreSQL has limited clustering support (later!)
- A clustered index is sometimes referred to as a **primary index**.
 - Can there be more than one clustered? Why?

Unclustered Indexes

- It is possible to create further indexes on a relation. Typical syntax:
 - CREATE INDEX myIndex ON involved(actorId);
- The unclustered indexes are sometimes called non-clustered or **secondary indexes**.
- Unclustered indexes:
 - Make most point queries more efficient.
 - Make some (narrow) range queries more efficient.

Clustered vs. Unclustered Index

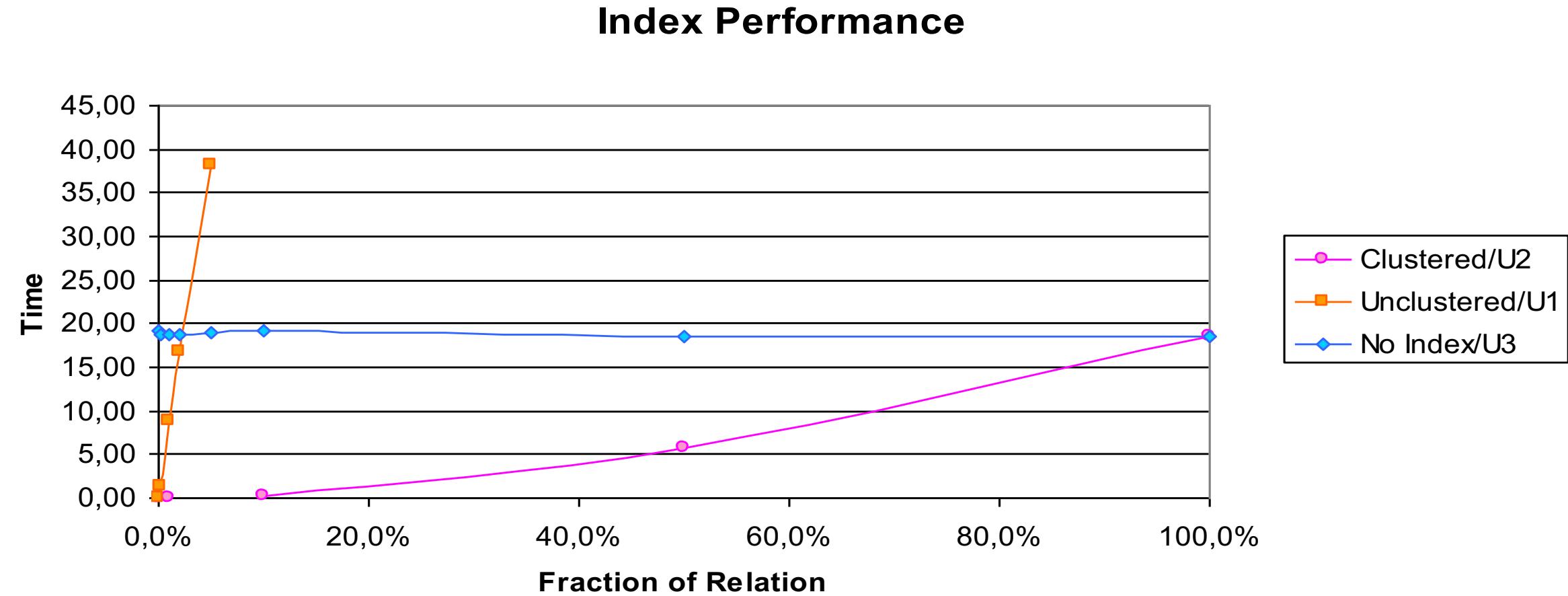
- To retrieve M records, where M is small:
 - Clustered: Probably one disk read
 - Unclustered: Probably M random disk reads
- To retrieve M records, where M is large:
 - Clustered: Probably $M/\text{records_per_page}$ sequential disk reads
 - Unclustered: Up to M random disk reads
- We still need to read the index itself – same for both!



Index Scan vs Full Table Scan

- Point and range queries on the attribute(s) of the clustered index are almost always best performed using an index scan.
- Non-clustered indexes should only be used with high selectivity queries.
 - Old rule of thumb: a secondary index scan is faster than a full table scan for queries returning less than 1% of a relation.
 - New rule of thumb?

Impact of Clustering on Performance



Covering Index

- An index that contains **ALL** attributes used in a query is called **covering**
 - Resulting query plans are index-only

```
SELECT COUNT(*) FROM movie WHERE year=1948;  
CREATE INDEX movieyear ON movie(year);
```

```
SELECT name FROM person WHERE height=170;  
CREATE INDEX phn ON person(height, name);
```

- The data from the relation is not needed = no disk reads required to retrieve tuples
 - Should a **covering** index be **clustered** or **unclustered**?
 - What is what matters then?

Clustered vs. Covering Index

- **To retrieve M records, where M is small:**
 - Clustered: Probably one disk read
 - Covering: Definitely need 0 disk reads
- **To retrieve M records, where M is large:**
 - Clustered: Probably $M/\text{records_per_page}$ sequential disk reads
 - Covering: Definitely need 0 disk reads
- **We still need to read the index itself – same for both!**

Practice: Types of Indexes

- **For each of the following queries:**

- Which index would give the best plan?
- Would the index be covering?
- Would you prefer clustered or unclustered index?
- Based on these queries, if you could choose, which index should be clustered?

(Q1) SELECT * ... WHERE birthdate = '20-02-2002'

(Q2) SELECT * ... WHERE height < 170

(Q3) SELECT * ... WHERE ID = 4564

(Q4) SELECT AVG(birthdate) ...

Multi-Attribute Indexes

- Defining an index on several attributes:

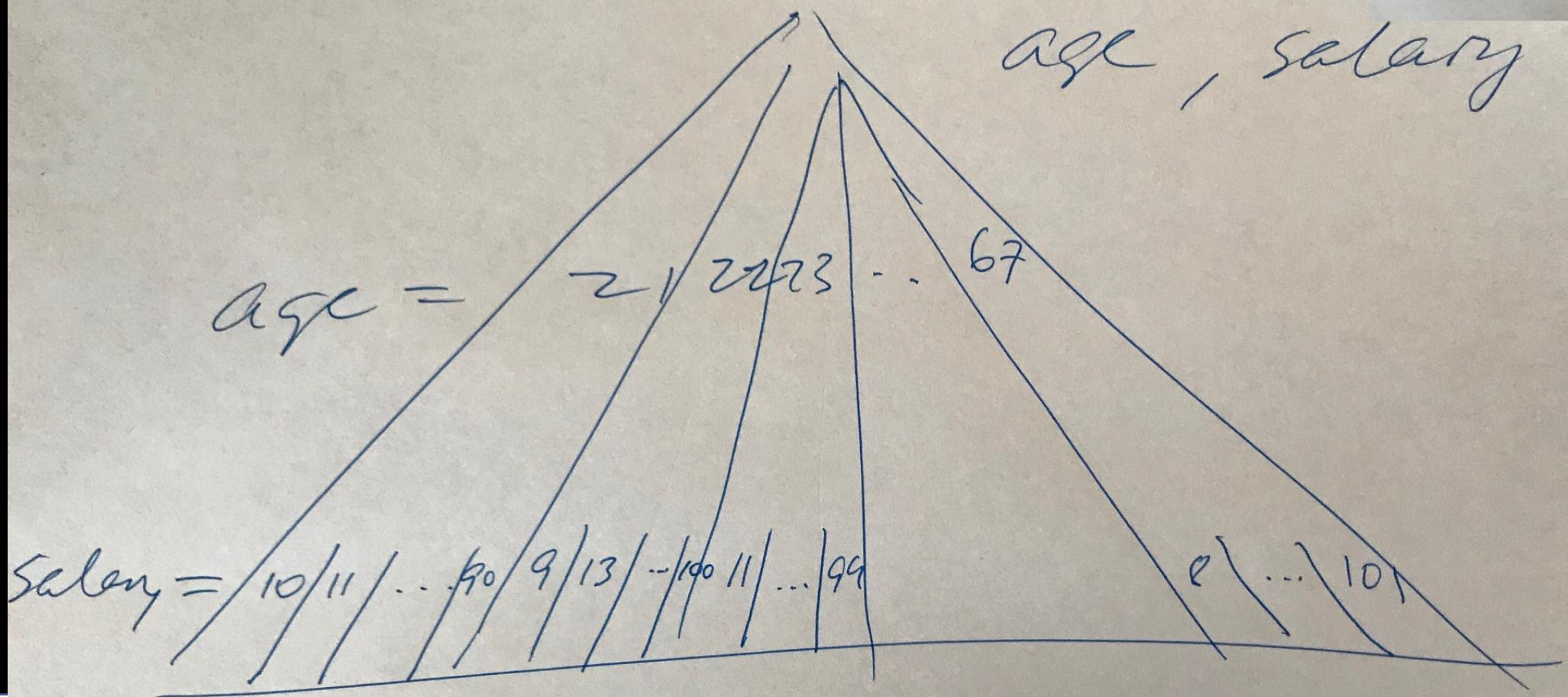
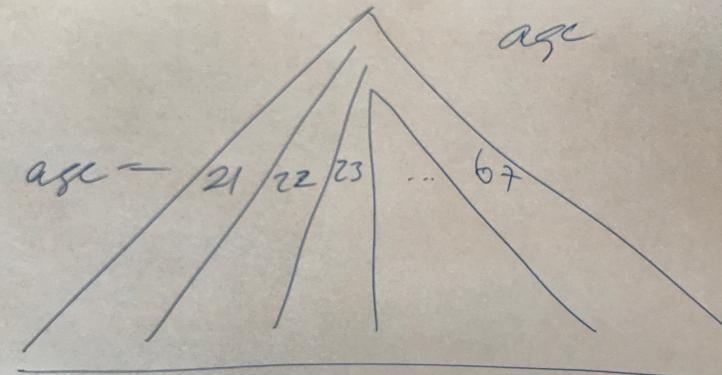
```
CREATE INDEX myIndex  
ON person (height,birthdate);
```

- Speeds up point/range queries such as:

```
SELECT *  
FROM person  
WHERE height=170 and birthdate<'1945-08-08'
```

- An index on several attributes usually gives index for any prefix of these attributes, due to lexicographic sorting.

Lexicographic Sorting?



Problem Session

- Which point and range queries are "easy" (equality on a prefix, range on one) when the relation is indexed with this two-attribute index?

```
CREATE INDEX myIndex  
ON person (height, birthdate);
```

- (Q1) A range query on height?
- (Q2) A range query on birthdate?
- (Q3) A point query on birthdate?
- (Q4) A point query on birthdate combined with a range query on height?
- (Q5) A point query on height combined with a range query on birthdate?

Choosing to Use an Index

- The choice of whether to use an index is made by the DBMS for every instance of a query
 - May depend on query parameters
 - You do not need to take indexes into account when writing queries
- Estimating selectivity is done using statistics
 - In PostgreSQL, statistics are gathered by executing statements such as `ANALYZE` involved

Choosing Columns

- **Candidates for index search keys**
 - Columns in WHERE clauses
 - Columns in GROUP BY clauses
 - Columns in ORDER BY clause

- **Columns that are rarely candidates**
 - Large columns (too much space)
 - Frequently updated columns (too much maintenance)
 - Columns in SELECT clauses (not used to find tuples)
 - ... but see covering indices!

What Speaks Against Indexing?

- **Space usage:**
 - Similar to size of indexed columns (plus pointer)
 - Most space for leaves, less for tree nodes
 - Not really important!
- **Time usage for keeping indexes updated under data insertion/change:**
 - Depends on the index architecture
 - This is important!

Other Impact of Indexes

The DBMS may use indexes in other situations than a simple point or range query.

- Some joins can be executed using a modest number of index lookups
 - May be faster than looking at all data
 - But hash-based joins are usually fastest (next lecture!)
- Some queries may be executed by only looking at the information in the index
 - Index only query execution plan ("covering index")
 - May need to read much less data.
- Consistency (checking keys and foreign keys)

Practice: Problem Session

- **What would be good indexes for this query?**

```
SELECT firstNames  
FROM person  
WHERE gender='m'  
AND firstnames LIKE 'Maria%';
```

Index types

- **Common:**

- B-trees (point queries, range queries)
- Hash tables (only point queries on the whole search key, but somewhat faster)
- Bitmap indexes (good for "dense" sets)

- **More exotic:**

- Full text indexes (substring searches)
- Spatial indexes (proximity search, 2D range search, multimedia, ...)
- ... and way more!

B+-tree Implementations in Some Major DBMS

- **Postgres**

- Cannot specify a clustered index!
- Manual CLUSTER command!
- Is an index on SERIAL clustered?

- **SQL Server**

- Table stored in clustered index
- Primary keys can be unclustered
- Indexes maintained dynamically

- **DB2**

- Table stored in clustered index
- Explicit command for index reorganization

- **Oracle**

- No clustered index until 10g
- Index organized table (unique/clustered)
- Indexes maintained dynamically

- **MySQL**

- Primary key is clustered
- Table stored in clustered index
- Indexes maintained dynamically



Takeaways

- **Large databases need to be equipped with suitable indexes**
 - Need understanding of what indexes might help a given set of queries
 - ‘Key’ distinction: Clustered vs Unclustered
 - A detailed understanding of various index types is beyond the scope of this course



What is next?

- **Next Lecture:**
 - Understanding Query Processing

Introduction to Database Systems

I2DBS – Spring 2023

- Week 9:
- Indexes (Recall)
- Query Processing

Jorge-Arnulfo Quiané-Ruiz

Readings:

PDBM 13.1



Profile of the Week

David DeWitt

Inventor of Hybrid Hash Join

- 1948: USA
- 1976: PhD in CS from the University of Michigan
- 1976: founded the Wisconsin Database Group
- 1984: Invented Hybrid Hash Join
- 1995: SIGMOD Edgar F. Codd Innovations Award
- 2009: ACM Software System Award

The Dewitt Clause

End-user license agreement provision that prohibits researchers and scientists from explicitly using the names of their systems in academic papers.





*Maintain
& tune*



RDBMS

Database Administrator
(DBA)

Indexes (Recall)

Readings:

PDBM 12

Recap: Indexing

- **Indexes are data structures that facilitate access to data from disk**
 - ... if conditions are a prefix of indexed attributes
 - Clustered indexes store tuples that match a range condition together
 - Some queries can be answered looking only at the index (a covering index for query)
 - Indexes slow down updates and insertions
- **The choice of whether to use an index is made by the DBMS for every instance of a query**
 - May depend on query parameters
 - Don't have to name indexes when writing queries

Processing Simple Selections

- Point and range queries on the attribute(s) of a clustered index are almost always best performed using an index scan
- Unclustered indexes should only be used with high selectivity queries
- Exception: Covering index is good for any selectivity
- If no index exists, a full table scan is required!
- If no “good” index exists, a full table scan is preferred!

Processing Complex Selections

- We consider the conjunction ("and") of equality and range conditions.
- No relevant index: Full table scan
- One relevant index:
 - Highly selective: Use that index
 - If not: Full table scan
- Multiple relevant indexes:
 - One is highly selective: Use that index
 - No single condition matching an index is highly selective: Can “intersect” the returned sets

Using a Highly Selective Index

- **Basic idea:**
 - Retrieve all matching tuples (few)
 - Filter according to remaining conditions
- **If index is clustered or covering: Retrieving tuples is particularly efficient, and the index does not need to be highly selective.**

Using Several Less Selective Indexes

- **For several conditions C_1, C_2, \dots matched by indexes:**
 - Retrieve the addresses R_i of tuples matching C_i .
 - The addresses are in the index leaves!
 - Compute the intersection $R = R_1 \cap R_2 \cap \dots$
 - Retrieve the tuples in R from disk (in sorted order)
- **Remaining problem:**
 - How can we estimate the selectivity of a condition? Of a combination of conditions?
 - Use some stats and probabilistic assumptions...

Example

```
SELECT title  
FROM Movie  
WHERE year = 1990  
AND studioName = 'Disney';
```

- **Examples of strategies:**
 1. Make a scan of the whole relation.
 2. Find movies from 1990 using index, then filter.
 3. Find Disney movies using index, then filter.
 4. Combine two indexes to identify rows fulfilling both conditions.
 5. Use one composite index to find Disney movies from 1990.
 6. Find Disney movies from 1990 and their titles in a composite covering index.

Example - Variant 1

```
SELECT title  
FROM Movie  
WHERE year = 1990  
AND studioName = 'Disney';
```

Available Indexes:

- CREATE INDEX yearIdx
ON Movie(year)
- CREATE INDEX studIdx
ON Movie(studioName)

- **Examples of strategies:**
 1. Make a scan of the whole relation.
 2. Find movies from 1990 using index, then filter.
 3. Find Disney movies using index, then filter.
 4. Combine two indexes to identify rows fulfilling both conditions.
 5. Use one composite index to find Disney movies from 1990.
 6. Find Disney movies from 1990 and their titles in a composite covering index.

Which strategies are possible and
which index would be used?

Example - Variant 2

```
SELECT title  
FROM Movie  
WHERE year = 1990  
AND studioName = 'Disney';
```

Available Indexes:

- CREATE INDEX yearIdx
ON Movie(year)
- CREATE INDEX yearStudioIdx
ON Movie(year, studioName)

- **Examples of strategies:**
 1. Make a scan of the whole relation.
 2. Find movies from 1990 using index, then filter.
 3. Find Disney movies using index, then filter.
 4. Combine two indexes to identify rows fulfilling both conditions.
 5. Use one composite index to find Disney movies from 1990.
 6. Find Disney movies from 1990 and their titles in a composite covering index.

Which strategies are possible and
which index would be used?

Example - Variant 3

```
SELECT title  
FROM Movie  
WHERE year = 1990  
AND studioName = 'Disney';
```

Available Indexes:

- CREATE INDEX idIdx
ON Movie(id)
- CREATE INDEX titleIdx
ON Movie(title)

- **Examples of strategies:**
 1. Make a scan of the whole relation.
 2. Find movies from 1990 using index, then filter.
 3. Find Disney movies using index, then filter.
 4. Combine two indexes to identify rows fulfilling both conditions.
 5. Use one composite index to find Disney movies from 1990.
 6. Find Disney movies from 1990 and their titles in a composite covering index.

Which strategies are possible and
which index would be used?

Processing Complex Selections Revisited

- We have considered the conjunction ("and") of a number of equality and range conditions.
- What about disjunctive ("or") selections?
 - One full table scan
OR
 - Multiple "and" queries



*Maintain
& tune*



RDBMS

Database Administrator
(DBA)

Query Processing

Readings:

PDBM 12

Query Evaluation in a Nutshell

- SQL rewritten to (extended) relational algebra
- The building blocks in DBMS query evaluation are algorithms that implement relational algebra operations.
 - Join is the most important one!
- May be based on:
 - Reading everything / Sorting / Hashing
 - Using indexes can sometimes help!
- The DBMS optimizer knows the characteristics of each approach, and attempts to use the best one in a given setting

Join Evaluation in a Nutshell

- **Join is the most important operation!**
- **May be based on:**
 - Reading everything / Sorting / Hashing
 - Using indexes can sometimes help!
- **We consider a simple join:**
 $R \text{ JOIN } S \text{ ON } S.ID = R.ID$
 - Extends to more complex joins in a straightforward way

Nested Loops Join

- The following basic algorithm can be used for any join:

```
for each tuple in R
    for each tuple in S
        if r.ID = s.ID
            then output (r, s)
```
- If the join condition is complex/broad, sometimes this is the only/best choice
 $R \text{ JOIN } S \text{ ON } S.ID <> R.ID$
- See animation example on LearnIT

Role of Index in Nested Loops Join

- If there is an index that matches the join condition, the following algorithm can be considered:
For each tuple in R
use the index to locate matching tuples in S
- See animation example on LearnIT
- Good if $|R|$ is small compared to $|S|$
- If many tuples match each tuple, a clustered or covering index is preferable.

Example

```
SELECT *
FROM Movie M, Producer P
WHERE M.year=2015
AND P.birthdate<'1940-01-01'
AND M.producer = P.id;
```

- **Some possible strategies:**
 1. Use index to find 2015 tuples, use index to find matching tuples in Producer.
 2. Use index to find producers born before 1940, use index to find matching movies.
 3. NL join Movie and Producer, then filter.

Problem Session

- What would be good indexes for this query?

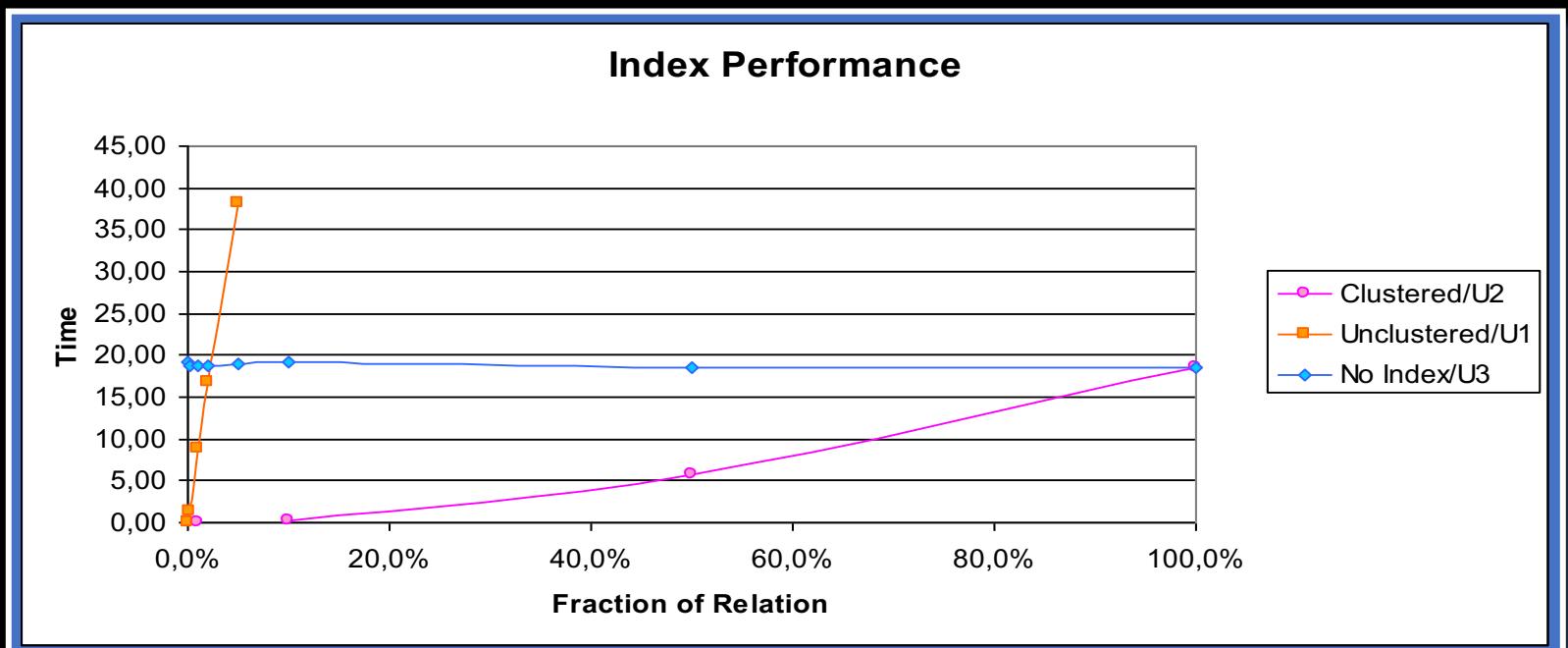
```
SELECT A.street, A.streetno  
FROM person P  
    JOIN address A ON A.person_id=P.id  
WHERE P.lastname='Bohr'  
AND P.firstnames LIKE 'Niels%';
```

Merge Join

- Consider R JOIN S on R.ID = S.ID
 - Step 0: Sort R and S on ID
 - Step 1: Merge the sorted R and S
 - See animation example on LearnIT
- Cost:
 - If already sorted: $O(|R| + |S|)$
 - Can we do better?
 - If not sorted:
 $O(|R|\log|R| + |S|\log|S| + |R| + |S|)$

Role of Indexes in Merge Joins

- Indexes can be used to read data in sorted order
- When is this a win?
 - Index is clustered
 - Index is covering
- When is this a loss?
 - Index is unclustered



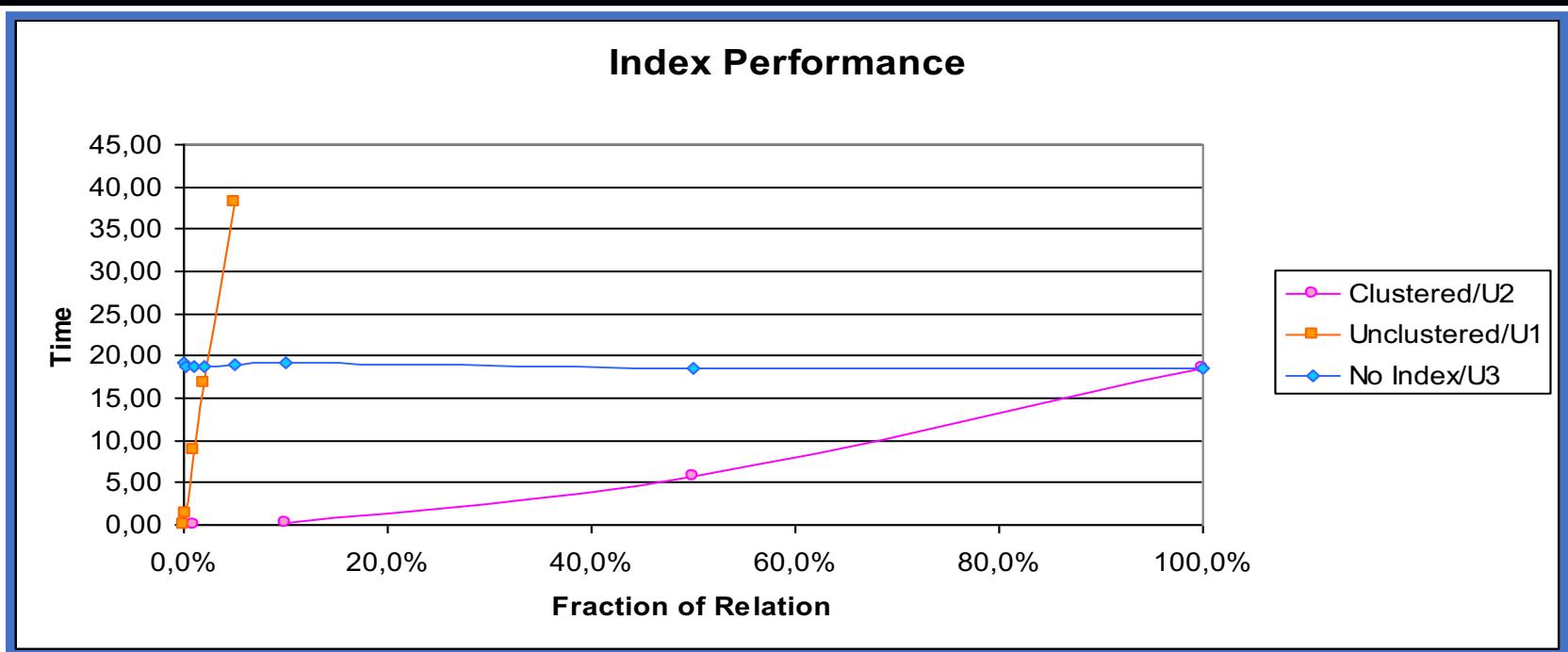
Hash Join

- Consider R JOIN S on R.ID = S.ID
 - Best if S fits in RAM

Step 0: Create a good hash function for ID
Step 1: Create a hash table for S in memory
Step 2: Scan R and look for matching tuples in the hash table
 - See animation example on LearnIT
- Cost: $O(|R| + |S|)$
 - Can we do better?
 - What if S does not fit in RAM?

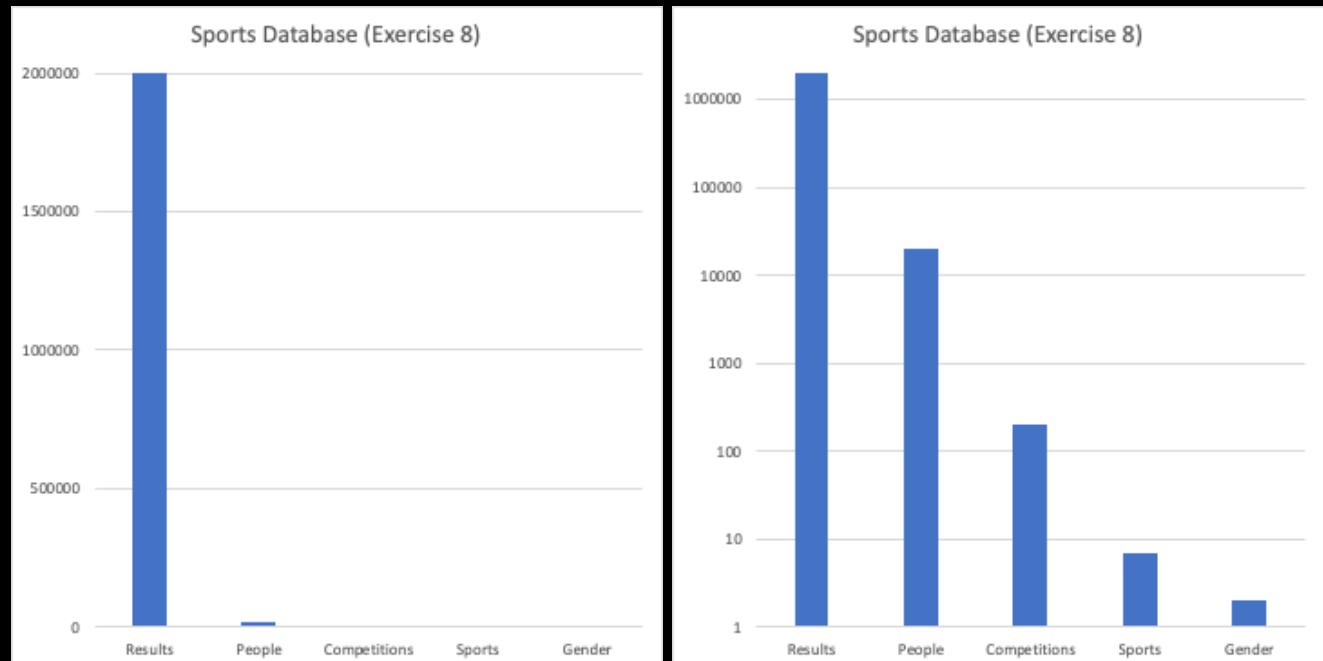
Role of Indexes in Hash Joins

- Hash joins read all the relations by default
- How can indexes be useful?
 - Apply to non-join conditions – before join
 - Index is covering



Comparison of Join Algorithms

- **Nested loops join:**
 - Very costly $O(IRI^*ISI)$
 - Works for any condition → sometimes only option
- **Merge join:**
 - Works well if data is well clustered
 - Works well if relations are large and similar in size
- **Hash join:**
 - Works well if one relation is small
 - Is that often the case?



Grouping Operations

- Many operations are based on grouping records (of one or more relations) according to the values of some attribute(s):
 - Join (group by join attributes)
 - Group by and aggregation (obvious)
 - Set operations (group by all attributes)
 - Duplicate elimination (group by all attributes)
- Most database systems implement such grouping efficiently using sorting or hashing

Partitioning of Tables

- **A table might be a performance bottleneck**
 - If it is heavily used, causing locking contention (more on this later in course)
 - If its index is deep (table has many rows or search key is wide), increasing I/O
 - If rows are wide, increasing I/O
- **Table partitioning might be a solution to this problem.**

Horizontal Partitioning

- If accesses are confined to disjoint subsets of rows, partition table into smaller tables containing the subsets
 - Geographically, organizationally, active/inactive
- Advantages:
 - Spreads users out and reduces contention
 - Rows in a typical result set are concentrated in fewer pages
- Disadvantages:
 - Added complexity
 - Difficult to handle queries over all tables

Vertical Partitioning

- Split columns into two subsets, replicate key
- Useful when table has many columns and
 - it is possible to distinguish between frequently and infrequently accessed columns
 - different queries use different subsets of columns
- Example: Employee table
 - Columns related to compensation (tax, benefits, salary) split from columns related to job (department, projects, skills).
- DBMS trend (for analytics):
 - Column stores, with full vertical partitioning.
 - More on this next week.



Takeaways

- **The performance difference between well-tuned and poorly-tuned applications can be massive!**
- **The DBMS does its best to optimize queries, but sometimes it needs help!**
 - Query tuning – rewrite as joins or non-correlated subqueries
 - Indexes – solve 90+% of all other performance problems
- **If that is not sufficient...**
 - Materialized views / Partitioning / Denormalization
 - Beyond the scope of this course!

What is next?

- 
- **Next Lecture:**
 - RDBMS Implementation
 - Main Memory DBMSes

Introduction to Database Systems

I2DBS – Spring 2023

- Week 10:
- RDBMS Implementation

Jorge-Arnulfo Quiané-Ruiz

Readings:

PDBM 2 and 14

Adapted slides from Eleni Tzirita
Zacharatou Björn Þór Jónsson

Some of the slides courtesy of
Pınar Tözün



Profile of the Week

Michael Stonebraker

A DBMS Pioneer

- 1943: Born in Newburyport, Massachusetts, USA
- 1971: PhD in CS from the University of Michigan
- Worked at UC Berkeley (until 2000) and MIT (now)
- 2014: Turing Award
- **Co-Inventor of:** Ingres, Postgres, Vertica, SciDB, ...



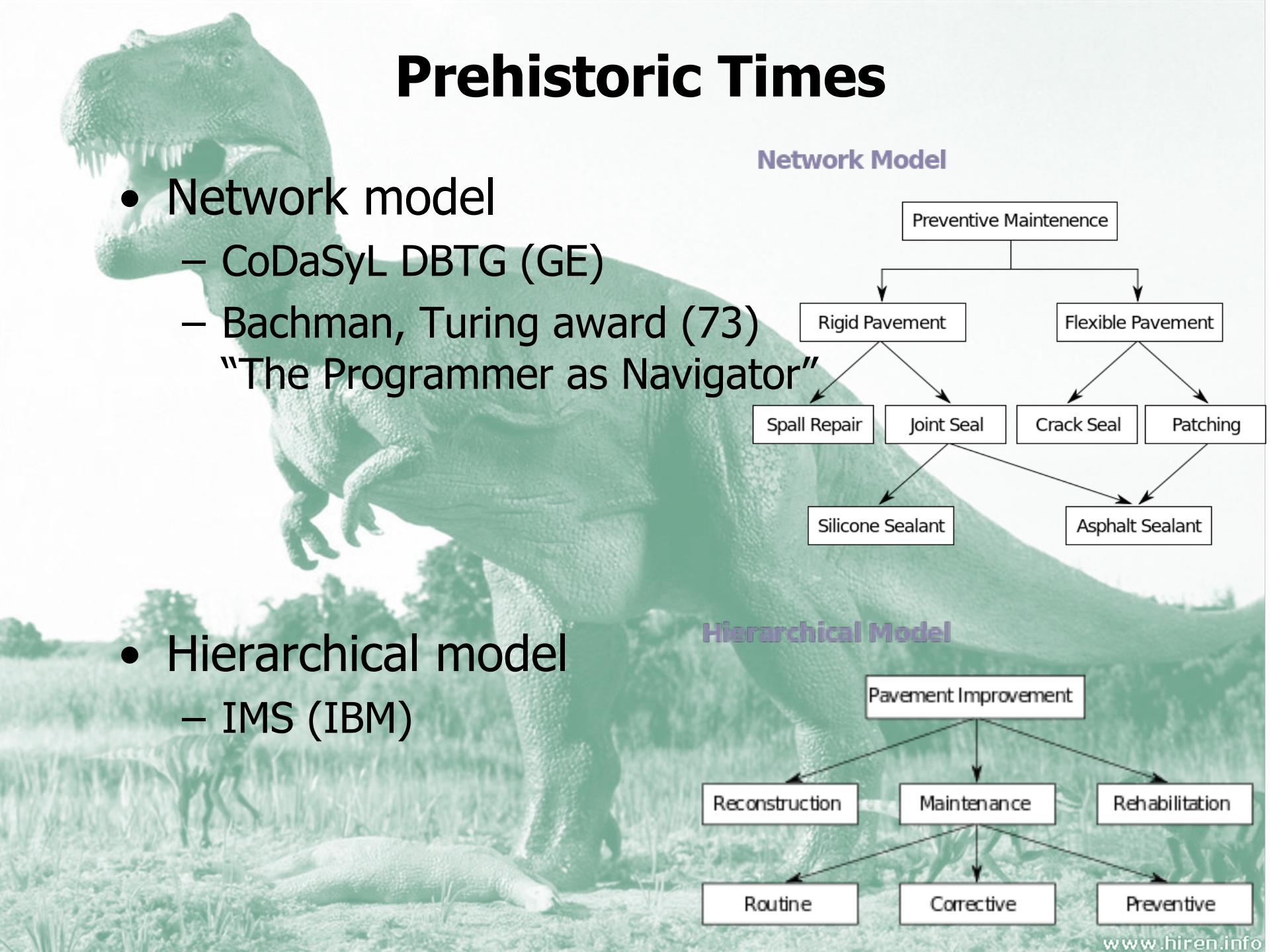


RDBMS

RDBMS Implementation

Readings:

PDBM 2

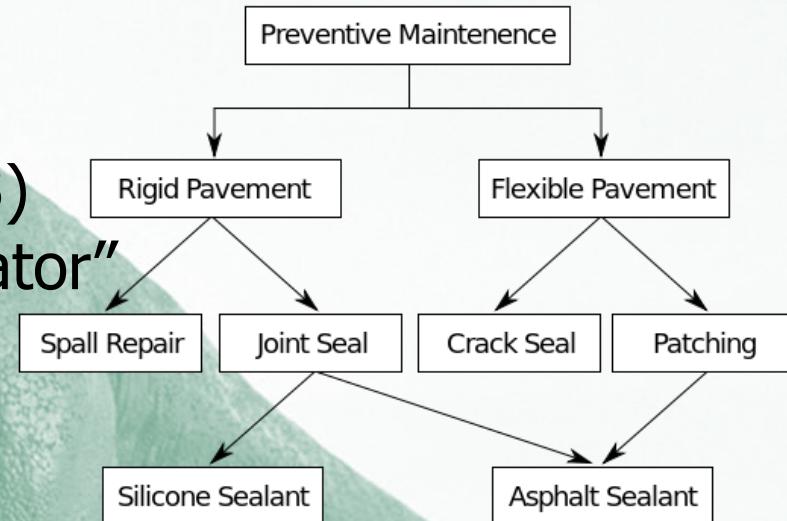


Prehistoric Times

- Network model
 - CoDaSyL DBTG (GE)
 - Bachman, Turing award (73)

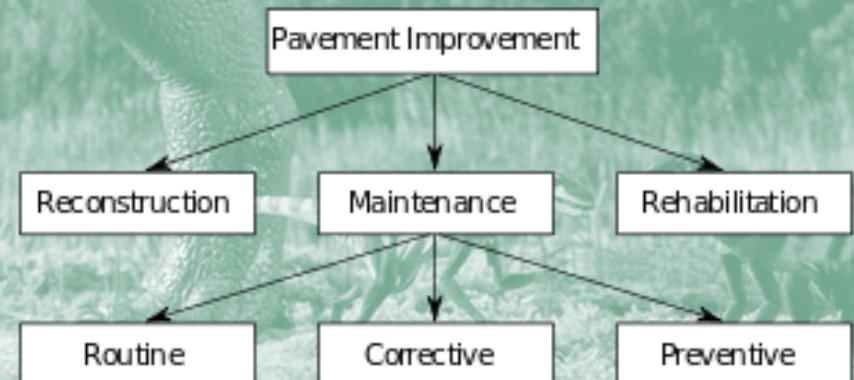
“The Programmer as Navigator”

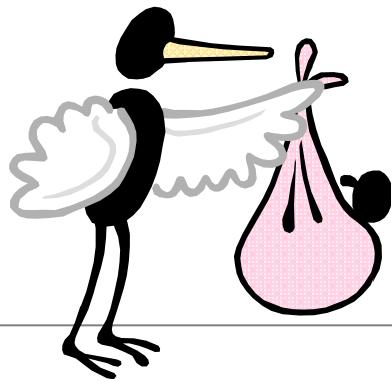
Network Model



- Hierarchical model
 - IMS (IBM)

Hierarchical Model





The Beginning

Information Retrieval

P. BAXENDALE, Editor

A Relational Model of Data for Large Shared Data Banks

E. F. CODD

IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] presently in vogue for non-inferential systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

A further advantage of the relational view is that it forms a sound basis for treating derivability, redundancy, and consistency of relations—these are discussed in Section 2. The network model, on the other hand, has spawned a number of confusions, not the least of which is mistaking the derivation of connections for the derivation of relations (see remarks in Section 2 on the “connection trap”).

Finally, the relational view permits a clearer evaluation

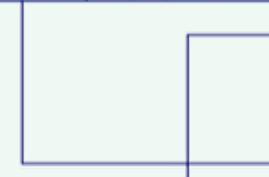
Problem with Existing Models

- Data and structure intertwined
 - “Queries” chase pointers
- Any change in structure invalidates programs

Now, consider the problem of printing out the part number, part name, and quantity committed for every part used in the project whose project name is “alpha.” The following observations may be made regardless of which available tree-oriented information system is selected to tackle this problem. If a program P is developed for this problem assuming one of the five structures above—that is, P makes no test to determine which structure is in effect—then P will fail on at least three of the remaining structures. More specifically, if P succeeds with structure 5, it will fail with all the others; if P succeeds with structure 3 or 4, it will fail with at least 1, 2, and 5; if P succeeds with 1 or 2, it will fail with at least 3, 4, and 5. The reason is simple in each case. In the absence of a test to determine which structure is in effect, P fails because an attempt is made to execute a reference to a nonexistent file (available systems treat this as an error) or no attempt is made to execute a reference to a file containing needed information. The reader who is not convinced should develop sample programs for this simple problem.

Since, in general, it is not practical to develop application programs which test for all tree structurings permitted by the system, these programs fail when a change in structure becomes necessary.

03-4472822	Random House	123 4th Street, New York
04-7733903	Wiley and Sons	45 Lincoln Blvd, Chicago
03-4859223	O'Reilly Press	77 Boston Ave, Cambridge
03-3920886	City Lights Books	99 Market, San Francisco



AuthorID	Auth...
345-28-2938	Haile S...
392-48-9965	Joe Bla...
454-22-4012	Sally H...
663-59-1254	Hanna...

ISBN	AuthorID	PubID	Date	Col...
1-34532-482-1	345-28-2938	03-4472822	1990	Col...
1-38482-995-1	392-48-9965	04-7733903	1985	Ma...
2-35921-499-4	454-22-4012	03-4859223	1952	Flu...
1-38278-293-4	663-59-1254	03-3920886	1967	Bea...

Relational Model

- Simple mathematical model
- Natural *declarative* query languages
 - Calculus → SQL
 - Algebra → Internal representation

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.

(Chamberlin, SQL Reunion)

Conflict?

- Hierarchical model
 - IMS (IBM)

Within IBM, the trouble was the existing database product, IMS. The company had already invested, both financially and organizationally, in the infrastructure and expertise required to sell and support it. A radical new technology had a great deal to prove before it could displace a successful, reliable, revenue-generating product such as IMS. Initially, the threat was minimal; Codd published his original paper in the open literature because no one at IBM (himself included) recognized its eventual impact. The response to this publication from the outside technical community, however, soon showed the company that the idea had great commercial potential. To head off this eventuality, IBM quickly declared IMS its sole strategic product, thus setting up Codd and his work to be criticized as counter to company goals.

Funding a Revolution

Information Retrieval

A Relational Model of Data for Large Shared Data Banks

E. F. CODD

IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected having to know how the data is organized in the machine internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed, and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and other facilities used by the user.

The Next Years

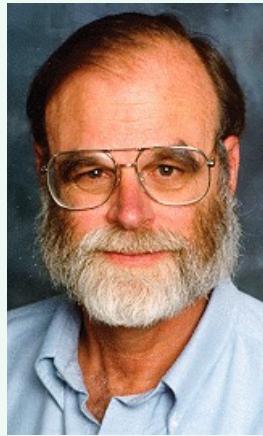
- Debates: Comparison to older models
 - + Types and simplicity of queries
 - ÷ Performance!? No “system”!
- Research projects
 - Ingres (Berkeley, 1973-1980+)
 - System R (IBM, 1974-1980+)



Major Milestones



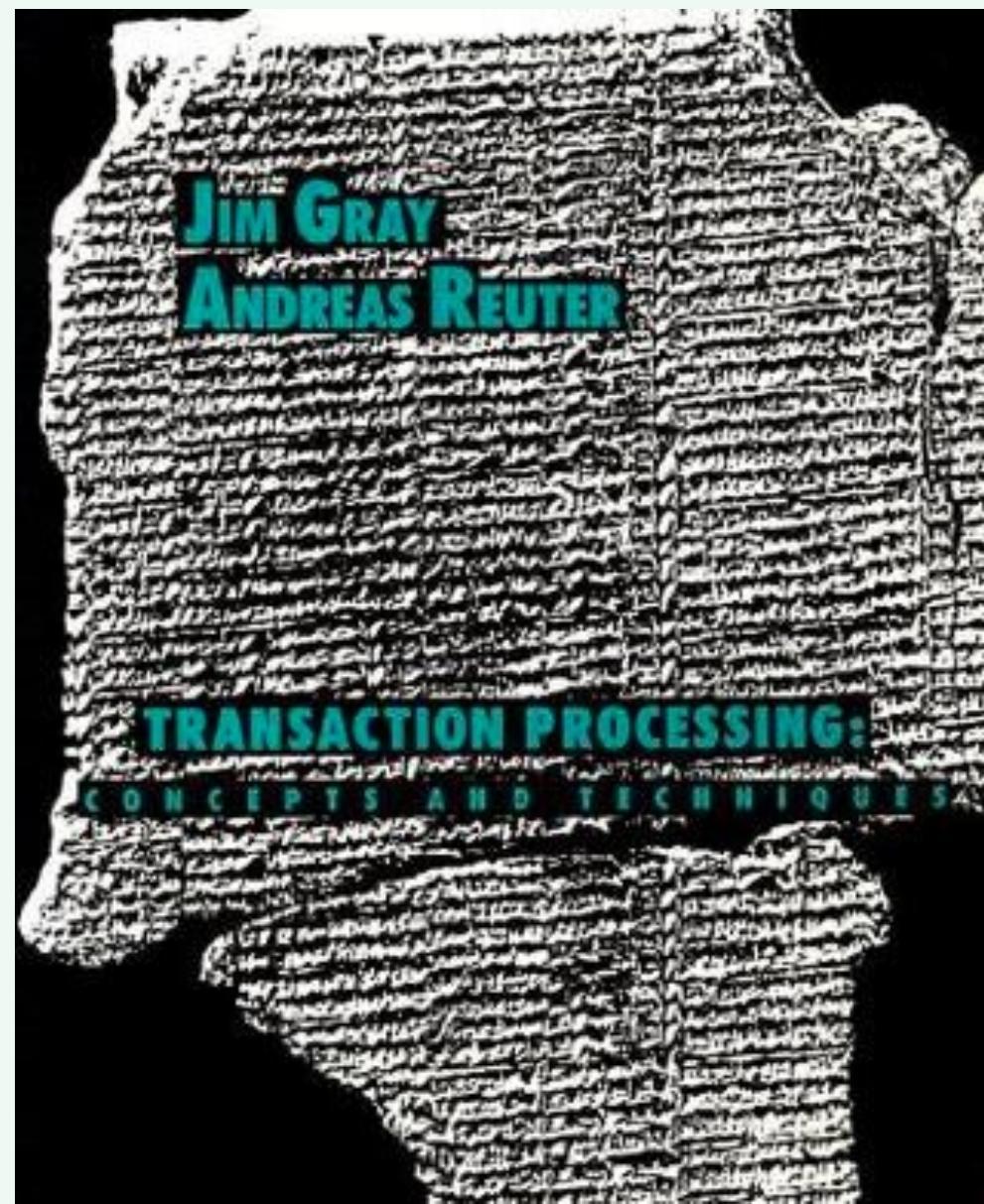
- Model (1970) Codd, Turing award (81)
 - SQL (1973-74) Chamberlin
 - Transactions (1975+) Gray, Turing award (98)
 - B+ trees (1977) Bayer, SIGMOD award
 - Logging (~1971)
 - Locking (1976) Gray et al.
 - Query optimization (1979) Selinger et al.
-
- Hash join (1986) Shapiro
 - ARIES recovery (1989) Mohan



Lost at Sea

From Jim Gray's first transaction to his final

DBMS implementation: he wrote “the book”



DBMS Architecture

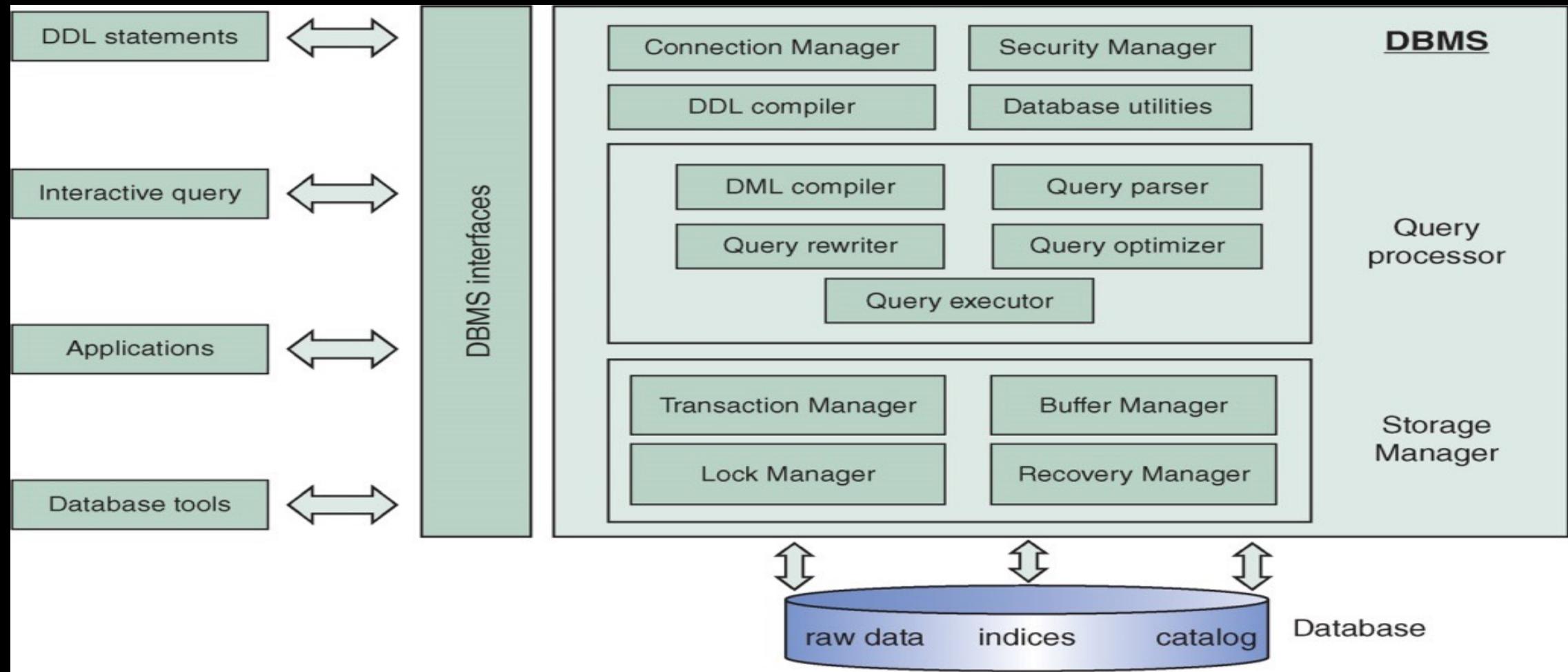


Figure 2.1

Why Do We Need Transactions?

Consider the following three transactions on the relation accounts (no, balance, type):

Transaction A

```
UPDATE accounts SET balance=balance*1.02 WHERE type='savings';
UPDATE accounts SET balance=balance*1.01 WHERE type='salary' AND balance>0;
UPDATE accounts SET balance=balance*1.07 WHERE type='salary' AND balance<0;
```

Transaction B

```
UPDATE accounts SET type='salary' WHERE no=12345;
```

Transaction C

```
UPDATE accounts SET balance=balance-1000 WHERE no=12345;
```

Assume that account 12345 starts as 'savings' with balance=500. What are the possible balance values after running transactions A, B and C?

Why Do We Need Transactions?

Consider the following three transactions on the relation accounts (no, balance, type):

Transaction A

```
UPDATE accounts SET balance=balance*1.02 WHERE type='savings';
UPDATE accounts SET balance=balance*1.01 WHERE type='salary' AND balance>0;
UPDATE accounts SET balance=balance*1.07 WHERE type='salary' AND balance<0;
```

Transaction B

```
UPDATE accounts SET type='salary' WHERE no=12345;
```

Transaction C

```
UPDATE accounts SET balance=balance-1000 WHERE no=12345;
```

Assume that account 12345 starts as 'savings' with balance=500.

A→B→C: -490

A.1: 510

A→C→B: -490

B: 510 (salary)

B→A→C: -495

A.2: 515.10

B→C→A: -535

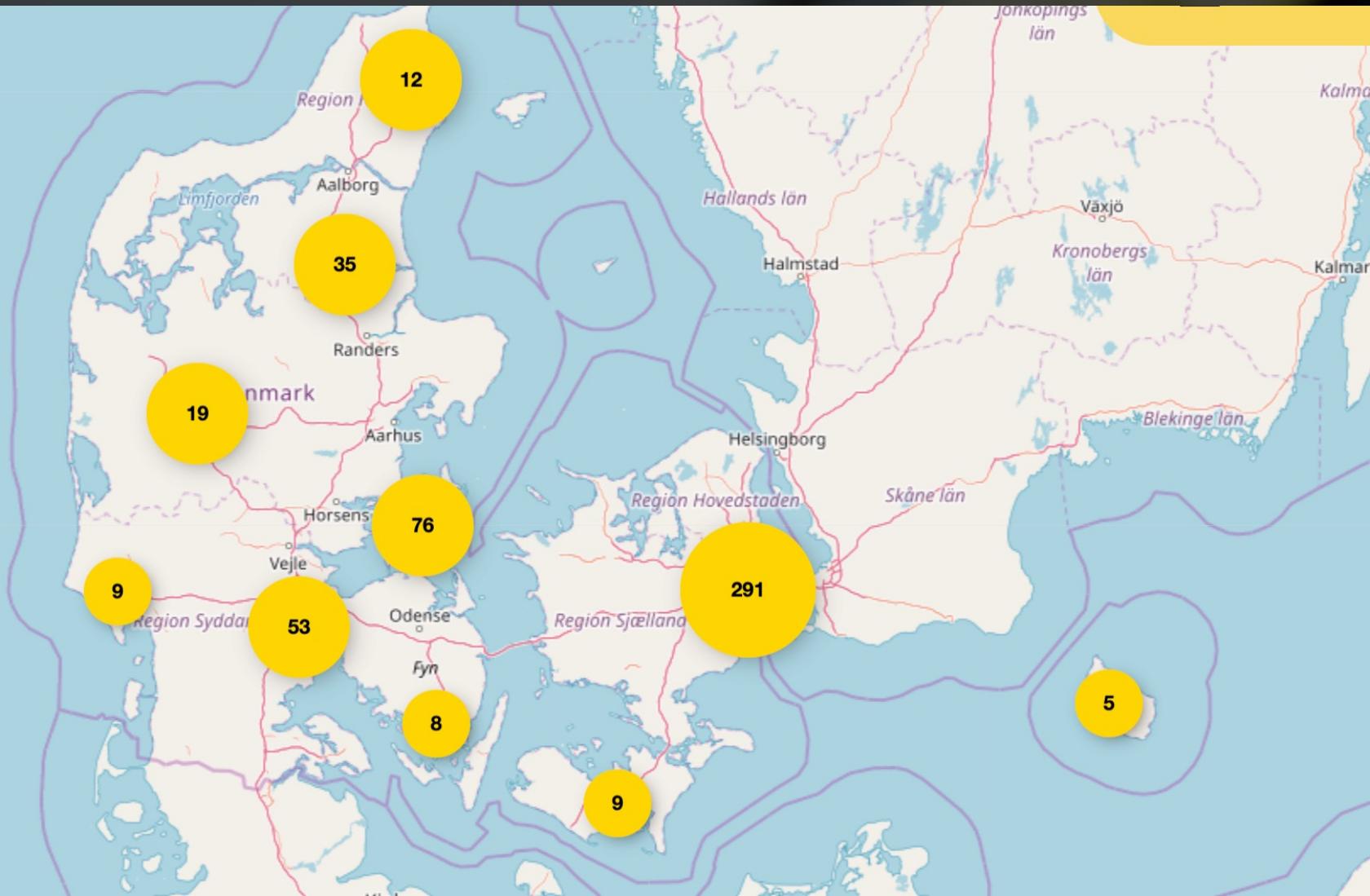
C: -484.90

C→A→B: -510

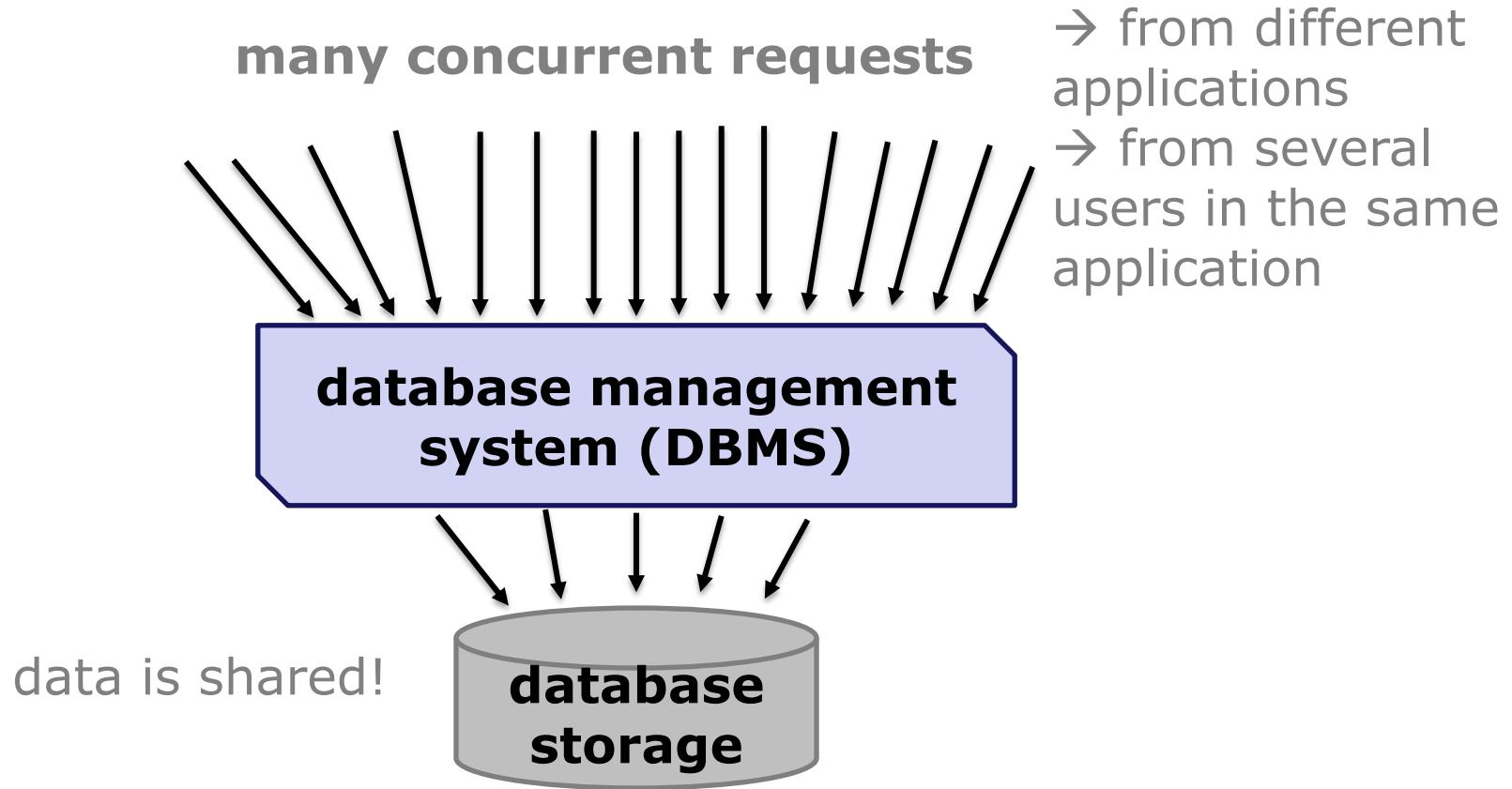
A.3: -518.843

C→B→A: -535

Why Do We Need Transactions?

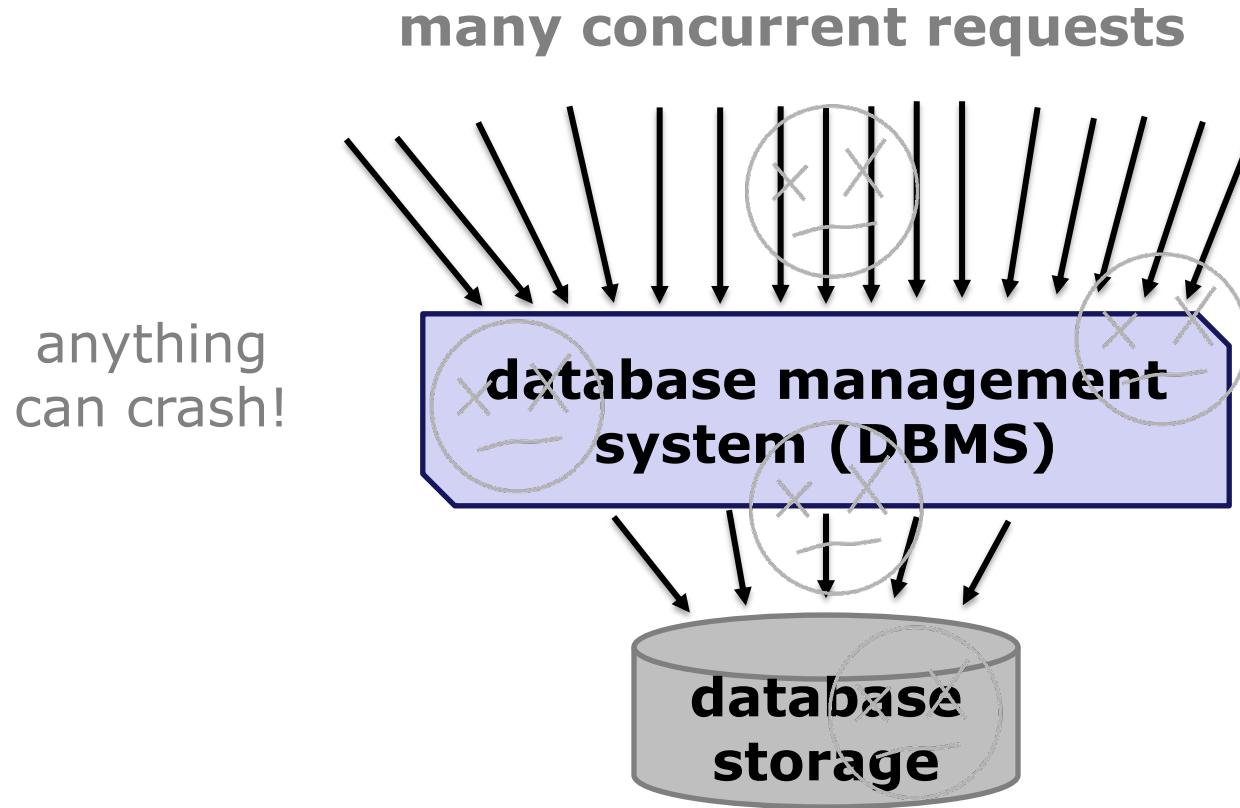


Why Do We Need Transactions?



**DBMS must ensure reliable operations over shared data
despite many concurrent accesses**

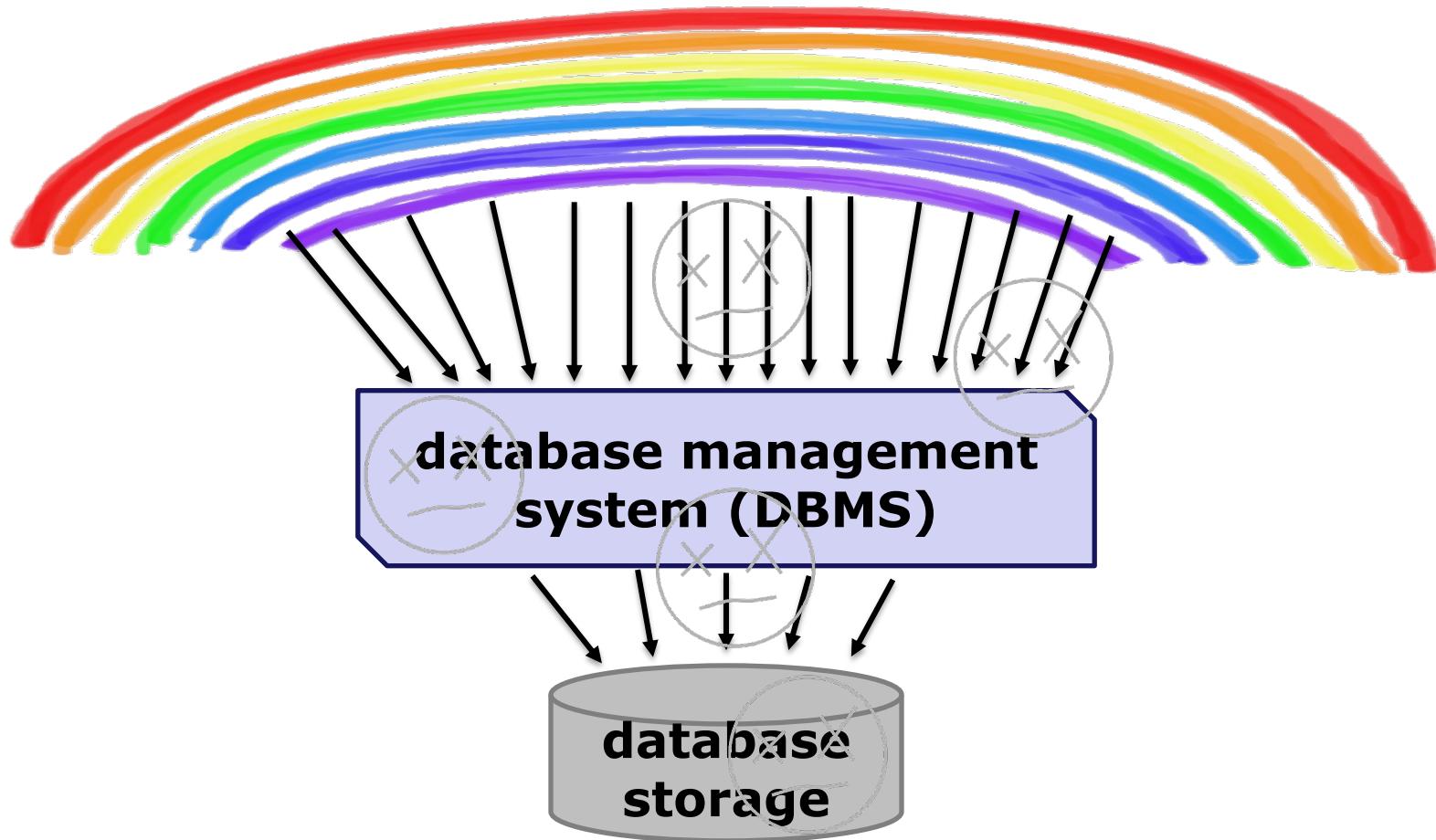
Why Do We Need Transactions?



DBMS must be resilient despite many potential system failures



**to the end user everything looks fine,
thanks to transactions!**



ACID Properties of Transactions

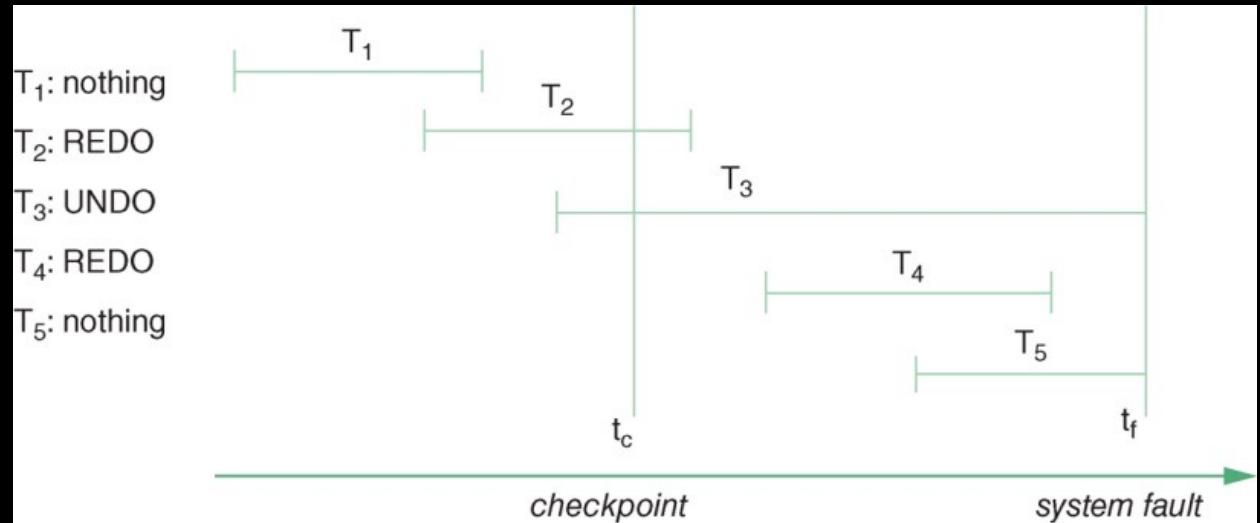
- **Atomicity:** Each transaction runs to completion or has no effect at all.
- **Consistency:** After a transaction completes, the integrity constraints are satisfied.
- **Isolation:** Transactions executed in parallel have the same effect as if they were executed sequentially.
- **Durability:** The effect of a committed transaction remains in the database even if the computer crashes.

How to Implement Transactions?

- Consistency \approx satisfying constraints
 - Use indexes for primary and foreign keys, ...
- Isolation = preventing corrupting changes
 - Pessimistic: Locking to prevent conflicts
 - Optimistic: Time stamps to detect conflicts
- Atomicity and durability = tracking changes
 - Logging “before” values to undo changes
 - Logging “after” values to redo changes

Atomicity and Durability Issues

- Atomicity
 - Transactions abort
 - Systems crash
- Durability
 - Systems crash
- Upon restart
 - Want to *see* effects of T1, T2, T4
 - Want to *remove* the effects of T3, T5

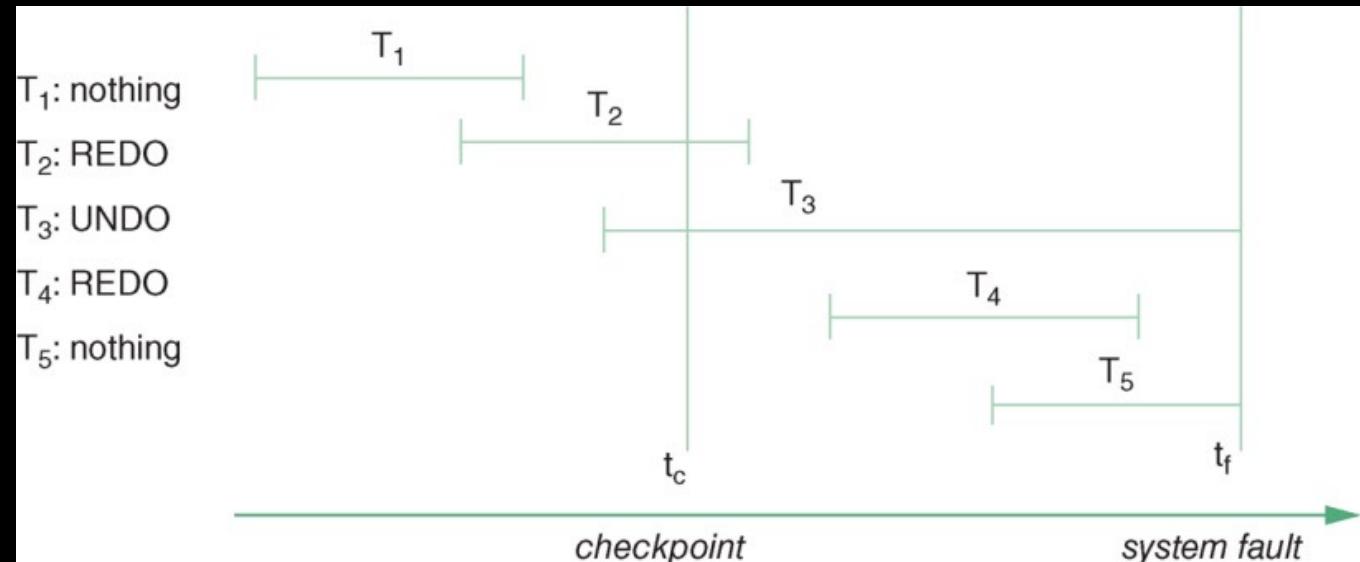


Buffer Management (RAM vs. disk)

- FORCE = write changed pages to disk at COMMIT
 - Book: „immediate update“ policy
 - Ensures Durability (assuming writes are atomic)
 - Increases response time
 - Will we FORCE changes to disk at COMMIT? NO
- STEAL = allow updated pages to be replaced
 - Book: NO STEAL = “deferred update“ policy
 - Ensures Atomicity (can simply discard at abort)
 - Increases response time
 - Will we guarantee NO STEAL of dirty pages? NO

STEAL/NO FORCE

- STEAL: Changes to disk before COMMIT
 - What if transaction aborts? system crashes?
 - Need to remember the *old* value of P to be able to *UNDO* the changes
- NO FORCE: Changes in RAM after COMMIT
 - What if system crashes?
 - Need to remember the *new* value of P to be able to *REDO* the changes



WAL Protocol

- Write-Ahead Logging

1. Before any **changes are written to disk** we force **the corresponding log record** to disk
2. Before **a transaction is committed** we force **all log records for the transaction** to disk

#1 ensures Atomicity

#2 ensures Durability

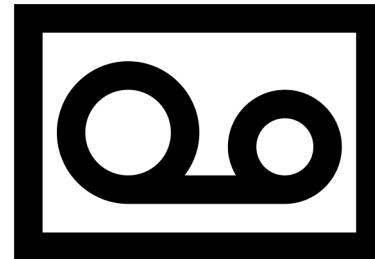
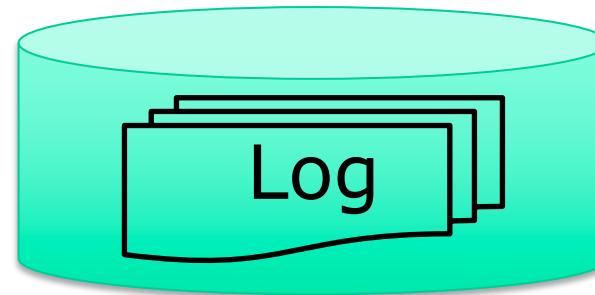
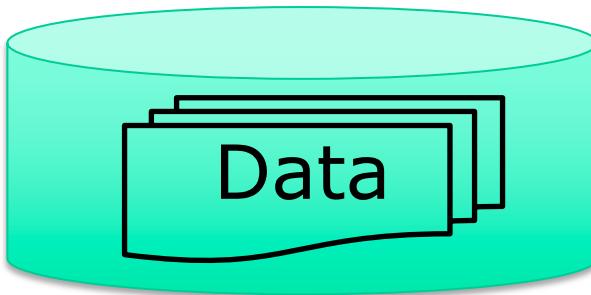
Key Concept: The Log

- Write REDO and UNDO info to log
 - Ordered list of REDO/UNDO info
 - Think of an infinite file with append only!
- Log processing must be fast – why? and how?
 - Write minimal info to log (diff)
 - <xid, pageID, offset, length, old_data, new_data> + control info
 - Many log entries per page
 - Ensure sequential writes!
 - Writing a log entry \Rightarrow writing all previous entries
 - **Put log on its own disk!**

Restart Recovery

1. Analyze information about transactions from the last checkpoint
2. REDO the changes of committed transactions that did not make it to disk
3. UNDO the changes of uncommitted transactions that accidentally made it to disk

Database Setup for DBAs

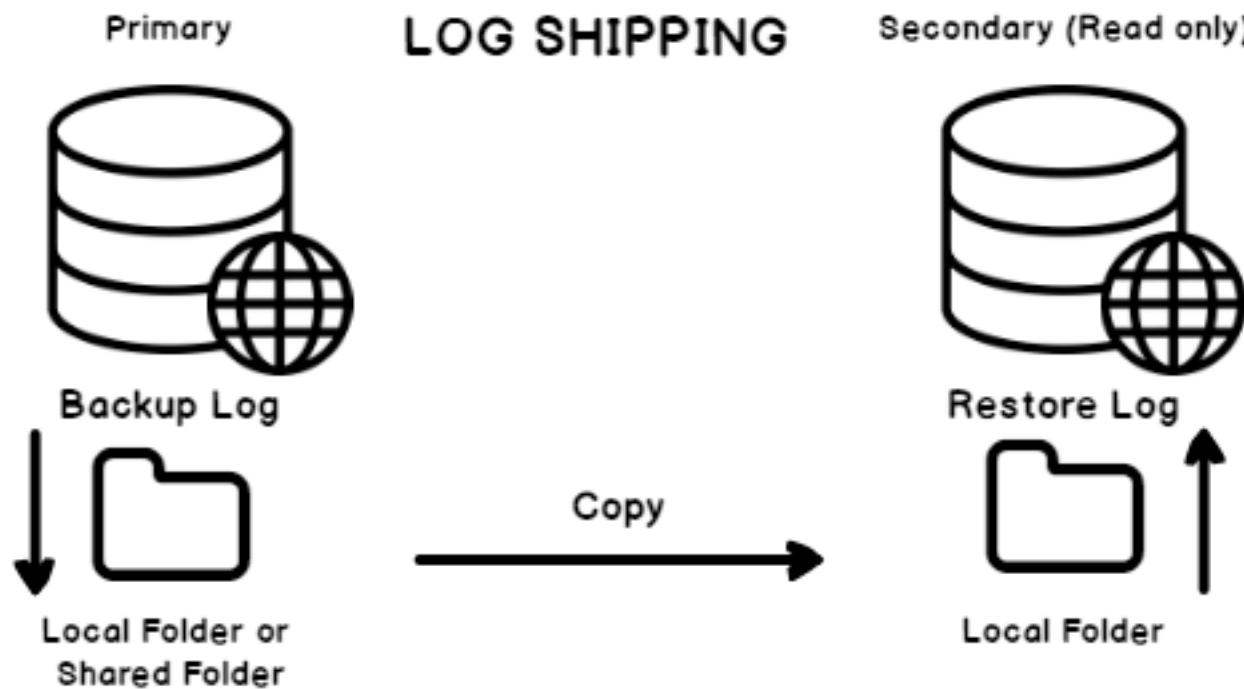


Backup

- On crash:
 - Data + Log = Recovery succeeds
 - Backup + Log = Recovery succeeds
 - Data + Backup (No Log) = Recovery FAILS!

High Availability?

- Typical approach: Second (failover) server
- Transaction log used to keep it up to date



How to Implement Transactions?

- Consistency \approx satisfying constraints
 - Use indexes for primary and foreign keys, ...
- Isolation = preventing corrupting changes
 - **Pessimistic: Locking to prevent conflicts**
 - Optimistic: Time stamps to detect conflicts
- Atomicity and durability = tracking changes
 - Logging “before” values to undo changes
 - Logging “after” values to redo changes

Isolation and Serializability

- Want transactions to satisfy *serializability*:
 - The state of the database should always look as if the committed transactions ran in some *serial schedule*.
- The scheduler of the DBMS is allowed to choose the order of transactions:
 - It is not necessarily the transaction that is started first, which is first in the serial schedule.

A Simple Scheduler

- A simple scheduler would maintain a queue of transactions and carry them out in order.
- Problems:
 - Transactions must *wait* for each other, even if unrelated (e.g. requesting data on different disks).
 - Some transactions may take very long, e.g. when external input or remote data is needed during the transaction.
 - Possibly smaller throughput. (Why?)

A Simple Scheduler

- A simple scheduler would maintain a queue of transactions and carry them out in order.
- Some believe this is fine for transaction processing, especially for main memory DBs:

The End of an Architectural Era (It's Time for a Complete Rewrite)

Michael Stonebraker
Samuel Madden
Daniel J. Abadi
Stavros Harizopoulos

Nabil Hachem
AvantGarde Consulting, LLC
nhachem@agdba.com

Pat Helland
Microsoft Corporation
phelland@microsoft.com

Inter-leaving Schedules

- Most DBMSs still have schedulers that allow the actions of transactions to interleave.
- However, the result should be **as if** some serial schedule was used.
- We will now study a mechanism that *enforces* "serializability": **Locking**.
- Other methods exist: Time stamping / optimistic concurrency control.
 - Out of scope for this course.

Locks

- In its simplest form, a lock is a right to perform operations on a database element.
- Only one transaction may hold a lock on an element at any time.
- Locks must be requested by transactions and granted by the locking scheduler.

Rigorous Two-Phase Locking

- Rigorous 2PL protocol:
 1. Before reading a record/page, get a shared (S) lock
Before writing a record/page, get a write (X) lock
 2. A record/page cannot have a write lock at the same time as any other lock
 3. Release all locks on COMMIT/ABORT
- This is commonly implemented, since:
 - Simple to understand and works well in practice
 - It makes transaction **rollback** easier to implement
- But:
Optimistic methods are growing in popularity!

Locks and Deadlocks

- The DBMS sometimes must make a transaction wait for another transaction to release a lock.
- This can lead to deadlock if e.g. A waits for B, and B waits for A.
- In general, we have a deadlock exactly when there is a cycle in the waits-for graph.
- Deadlocks are resolved by aborting some transaction involved in the cycle.

Avoiding Deadlocks

- Upgrade requests can also deadlock

```
SELECT :x=counter  
FROM table  
WHERE <condition>  
:x=:x+1  
  
UPDATE table  
SET counter = :x  
WHERE <condition>
```

```
UPDATE table  
SET counter = counter+1  
WHERE <condition>  
  
SELECT :x=counter  
FROM table  
WHERE <condition>
```

```
SELECT FOR UPDATE :x=counter  
FROM table  
WHERE <condition>  
:x=:x+1  
  
UPDATE table  
SET counter = :x  
WHERE <condition>
```

- Order matters
 - With consistent order of access deadlocks are avoided
 - Why B+-tree access does not deadlock
 - Optimizer may not allow control over order!

Phantom Tuples

- Suppose we lock tuples where $A=1$ in a relation, and subsequently another tuple with $A=1$ is inserted.
- For some transactions this may result in unserializable behaviour, i.e., it will be clear that the tuple was inserted during the course of a transaction.
- Such tuples are called phantoms.

Phantom Example

```
create table B (x int, y int);
```

```
insert into B values (1, 2);
```

```
begin;
```

```
select min(y) from B where x = 1;
```

```
begin;
```

```
insert into B values (1, 1);
```

```
commit;
```

-- Repeat the SAME read!

```
select min(y) from B where x = 1;
```

```
commit;
```

Avoiding Phantoms

- Phantoms can be avoided by putting an exclusive lock on a relation before adding tuples.
 - However, this gives poor concurrency.
- A technique called “index locking” can be used to prevent other transactions from inserting phantom tuples but allow most non-phantom insertions.
- In SQL, the programmer may choose to either allow phantoms in a transaction or insist they should not occur.

Isolation Levels in Modern Systems

- **READ UNCOMMITTED**
a transaction can read uncommitted changes
- **READ COMMITTED**
a transaction only reads committed data,
some other transaction may overwrite this data
- **REPEATABLE READ**
a transaction only reads committed data,
other transactions cannot overwrite this data,
but phantoms are possible
- **SERIALIZABLE**
ensures serializable schedule with no anomalies

may
violate
“I” in
ACID

ensures “I” in ACID always

figure taken from
"When is "ACID" ACID? Rarely."

Database	Default Isolation	Maximum Isolation
Actian Ingres 10.0/10S	S	S
Aerospike	RC	RC
Akiban Persistit	SI	SI
Clustrix CLX 4100	RR	?
Greenplum 4.1	RC	S
IBM DB2 10 for z/OS	CS	S
IBM Informix 11.50	Depends	RR
MySQL 5.6	RR	S
MemSQL 1b	RC	RC
MS SQL Server 2012	RC	S
NuoDB	CR	CR
Oracle 11g	RC	SI
Oracle Berkeley DB JE	S	S
Oracle Berkeley DB JE	RR	S
Postgres 9.2.2	RC	S
SAP HANA	RC	SI
ScaleDB 1.02	RC	RC
VoltDB	S	S
Legend	<i>RC: read committed, RR: repeatable read, S: serializability, SI: snapshot isolation, CS: cursor stability, CR: consistent read</i>	

the entire world doesn't run on ACID!

but an important part of it does so!

Powerful Abstraction



- “Without transactions a DBMS isn’t”
- “Not exactly rocket science”

Major Milestones



- Model (1970) Codd, Turing award (81)
- SQL (1973-74) Chamberlin

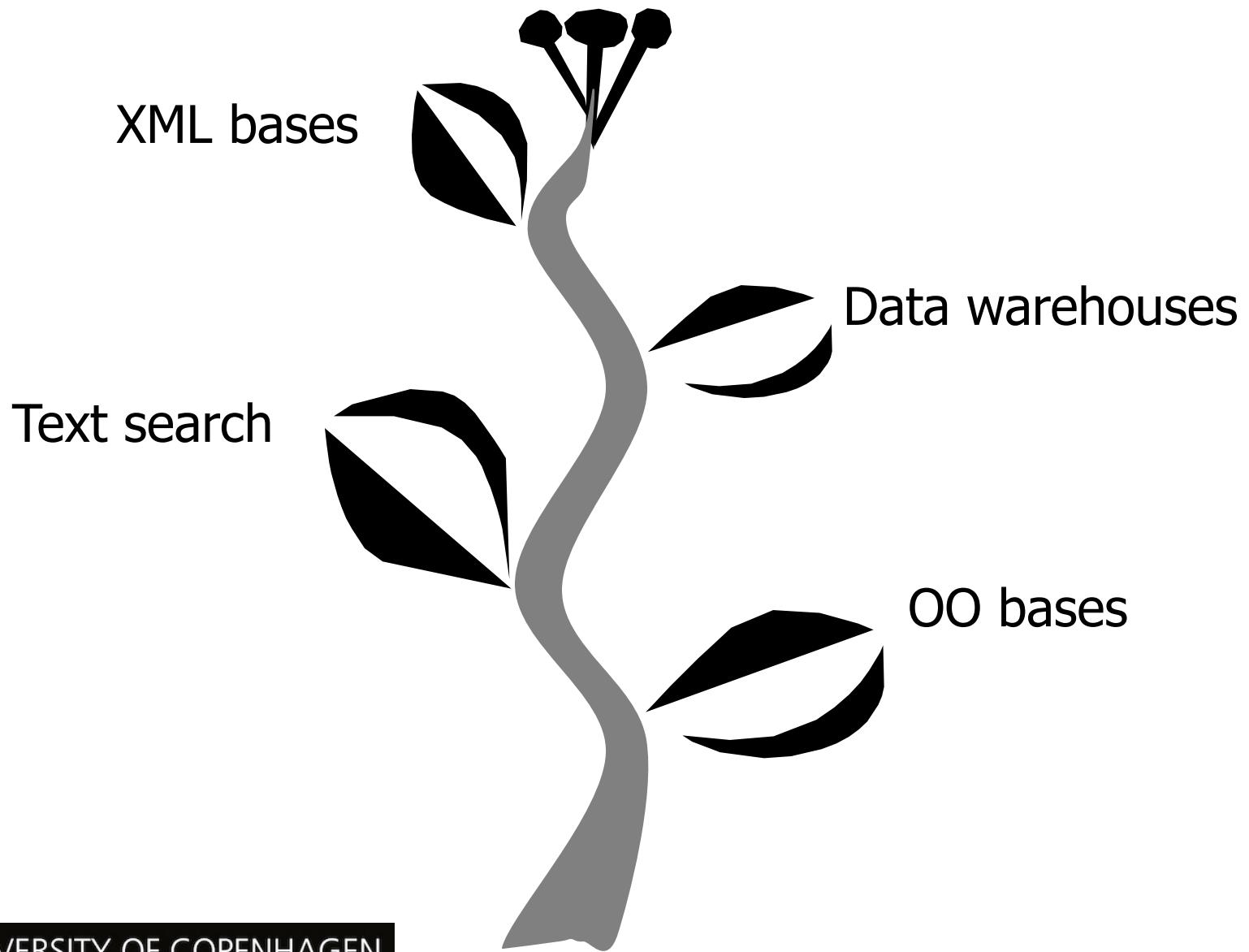
- Those relational database systems haven't changed for 30 years
anonymous DBA

- Query optimization (1979) Selinger et al.
- Hash join (1986) Shapiro
- ARIES recovery (1989) Mohan

The Internet Age → Changes!

- Users and queries
 - Cooperative systems; Query models; ...
- Architectures
 - SSDs; RAM; Distribution; Big data; Wireless; ...
- Data types
 - Media; Logs; Spatio-temporal; Sensor streams; ...

Evolution of Relational Systems





Takeaways

- **RDBMS implementation:**
 - Transactions: Locking and logging
 - Queries: SQL and query optimization
 - Based on old assumptions
- **Next two weeks:**
 - OLTP and OLAP require different implementation techniques:
Storing rows vs columns + optimizations...
 - MMDBMS Implementation makes better use of modern
hardware properties
 - NoSQL: Move even further from RDBMs
 - Big Data and Analytics: ... even further...



What is next?

- **Next Lecture:**
 - Scaling-out
 - NoSQL
 - Eventual consistency
 - CAP theorem

Introduction to Database Systems

I2DBS – Spring 2023

- Week 11:
- MMDBS
- Scaling Out
- NoSQL

Jorge-Arnulfo Quiané-Ruiz

Readings:

PDBM 11

Information

- **Homework 3:**
 - We will provide feedback and solution ASAP
- **Homework 4:**
 - It will be online this Friday April 28th, 2023
 - Deadline: May 12th, 2023
- **Trial Exam**
 - I will be open on May 8th and will close on May 12th , 2023
 - Once started, you have 4 hours to finish it
- **Final Exam**
 - It will have the same organization and structure as the trial and old exams
- **Course Anonymous Feedback**
 - It opens on May 1st, 2023



Profile of the Week

Martin L. Kersten

A DBMS Pioneer

- 1953: Born in Amsterdam, Netherlands
- 1985: PhD in CS from the Vrije Universiteit, Amsterdam
- 1985: he moved to Centrum Wiskunde & Informatica (CWI)
- 2014: SIGMOD Edgar F. Codd Innovations Award
- 2016: SIGMOD Systems Award
- **Co-Inventor of:** MonetDB, PRISMA, Data Cyclotron
- 2022: R.I.P.





RDBMS

Main Memory RDBMS

Early change: OLTP vs OLAP

- **OLTP = transactions**

- Short simple transactions
- Mostly INSERT or simple queries
- Gather data, no analysis
- Many users!

- **OLAP = analytics**

- Long complex queries
- Only SELECT, no inserts/updates
- Analysing gathered data
- Few (but heavy) users!

Row-Store vs Column-Store

- **OLTP = transactions**

- Access one/few rows
- Best to store whole rows
- This is traditional!

- **OLAP = analytics**

- Access whole tables
... but one/few columns
- Best to store columns together
- Significant performance improvements for large analytic queries

Recent change: Main Memory RDBMSs

- **Why did we need concurrent transactions?**
 - Needed OLAP-style queries (long, complex)
 - Needed to wait a long time for disk IOs since 1TB of memory was infeasible
- **What if OLTP only and all data is in RAM?**
 - All transactions are short
 - Never need to wait for disk
 - Note: There is still disk for persistent storage!
- **Do we need concurrent transactions?**
- **If not, how can we change the implementation?**

Main Memory RDBMSs



no disk use while executing tasks in a database system

(1) don't have to optimize data accesses for disk

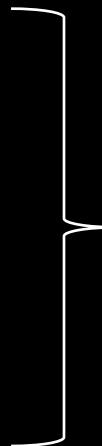
- no buffer manager
 - no need for tuples in slotted-pages in-memory, have direct access to tuples
 - indexes aren't kept in pages either & usually aren't persisted
- aim better cache utilization/accesses
 - row-store for transactions, column-store for analytics
 - cache-conscious index structures
 - query compilation that generates cache-conscious code

} also valid
for disk

Main Memory RDBMSs

- no disk use while executing tasks in a database system

(2) don't have disk IO to wait for anymore

- lightweight logging for recovery
 - need to access disk for redo logging
 - reduce log size (e.g., command logging)
 - non-blocking concurrency control
 - optimistic multi-versioning or
 - based on data partitioning
- 
- specific to transactions

Main Memory RDBMSs

- if we can also rely on short OLTP transactions...

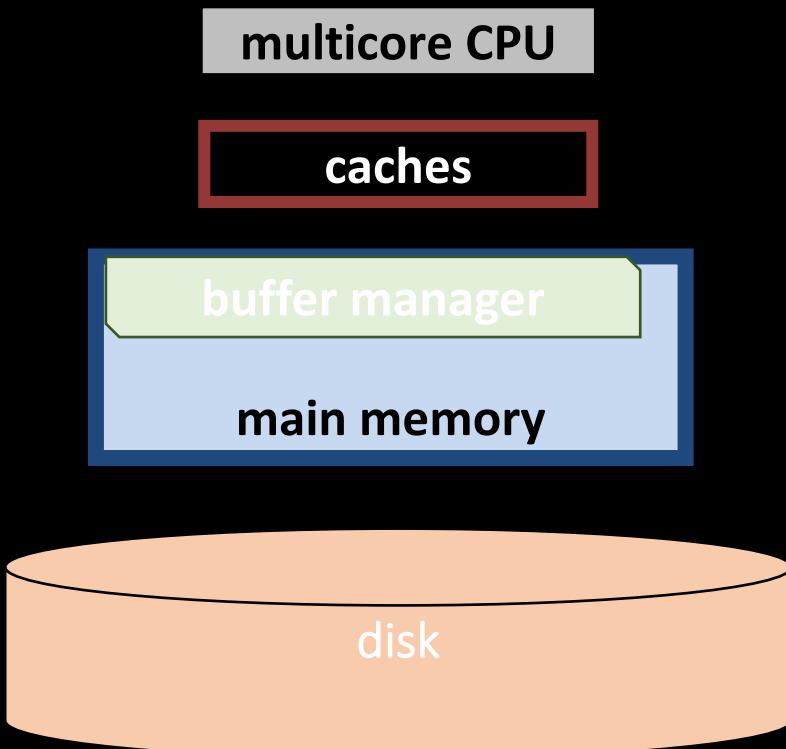
(3) don't have to wait for long transactions

- serial execution of transactions

} specific to
transactions

Main Memory OLTP System

Traditional system



main-memory OLTP system

- restricted/no concurrency control
- query compilation that generates more efficient code
- no buffer manager
- data organized for better cache utilization/accesses
- no disk use during transactions
- lightweight logging for recovery



RDBMS

Scaling Out

The End of an Architectural Era (It's Time for a Complete Rewrite)

Michael Stonebraker

Samuel Madden

Daniel J. Abadi

Stavros Harizopoulos

MIT CSAIL

{stonebraker, madden, dna,
stavros}@csail.mit.edu

Nabil Hachem

AvantGarde Consulting, LLC

nhachem@agdba.com

Pat Helland

Microsoft Corporation

phelland@microsoft.com

ABSTRACT

In previous papers [SC05, SBC+07], some of us predicted the end of “one size fits all” as a commercial relational DBMS paradigm. These papers presented reasons and experimental evidence that showed that the major RDBMS vendors can be outperformed by 1-2 orders of magnitude by specialized engines in the data warehouse, stream processing, text, and scientific database markets.

Assuming that specialized engines dominate these markets over me, the current relational DBMS code lines will be left with the business data processing (OLTP) market and hybrid markets where more than one kind of capability is required. In this paper we show that current RDBMSs can be beaten by nearly two orders of magnitude in the OLTP market as well. The experimental evidence comes from comparing a new OLTP engine, H-Store, which we have built at M.I.T., to a popular RDBMS on the standard transactional benchmark, TPC-C.

We find that the current RDBMS code lines, while they may be a “one size fits all” solution, in fact, excel at nothing. They are 25 year old legacy code lines that should be replaced by a collection of “from scratch” specialized engines (and the research community) designed specifically for their application domains.

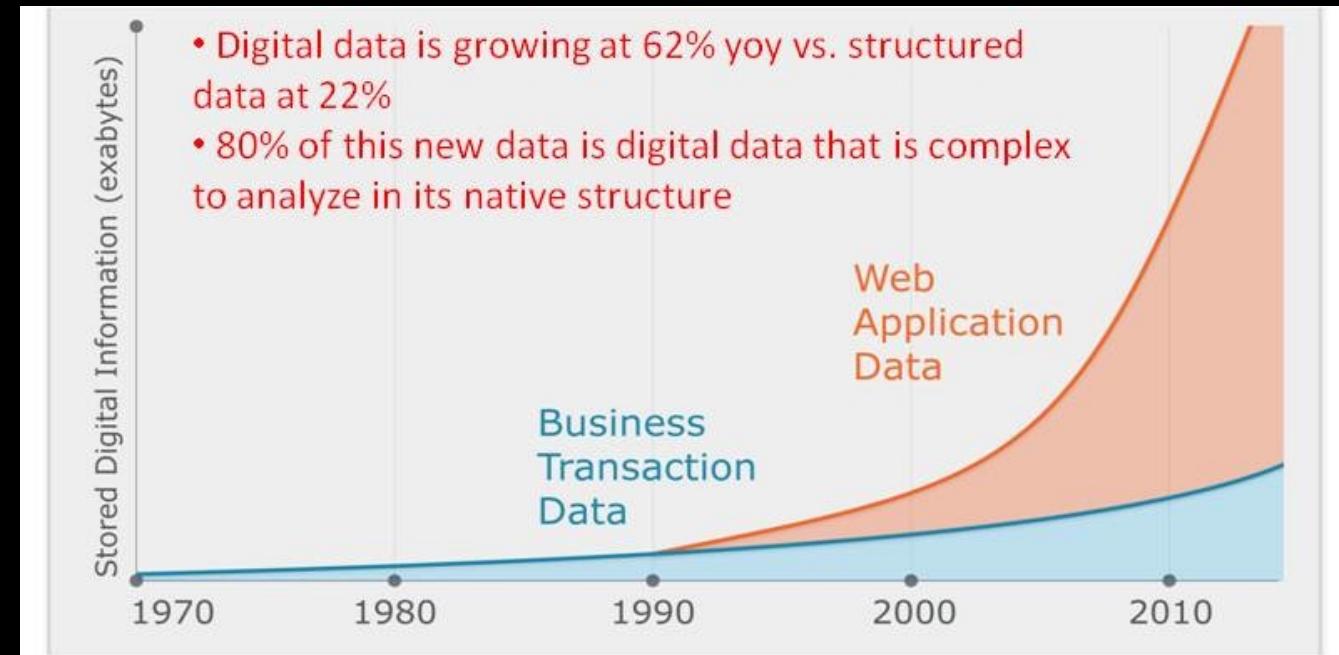
All three systems were architected more than 25 years ago, when hardware characteristics were much different than today. Processors are thousands of times faster and memories are thousands of times larger. Disk volumes have increased enormously, making it possible to keep essentially everything, if one chooses to. However, the bandwidth between disk and main memory has increased much more slowly. One would expect this relentless pace of technology to have changed the architecture of database systems dramatically over the last quarter of a century, but surprisingly the architecture of most DBMSs is essentially identical to that of System R.

Moreover, at the time relational DBMSs were conceived, there was only a single DBMS market, business data processing. In the last 25 years, a number of other markets have evolved, including data warehouses, text management, and stream processing. These markets have very different requirements than business data processing.

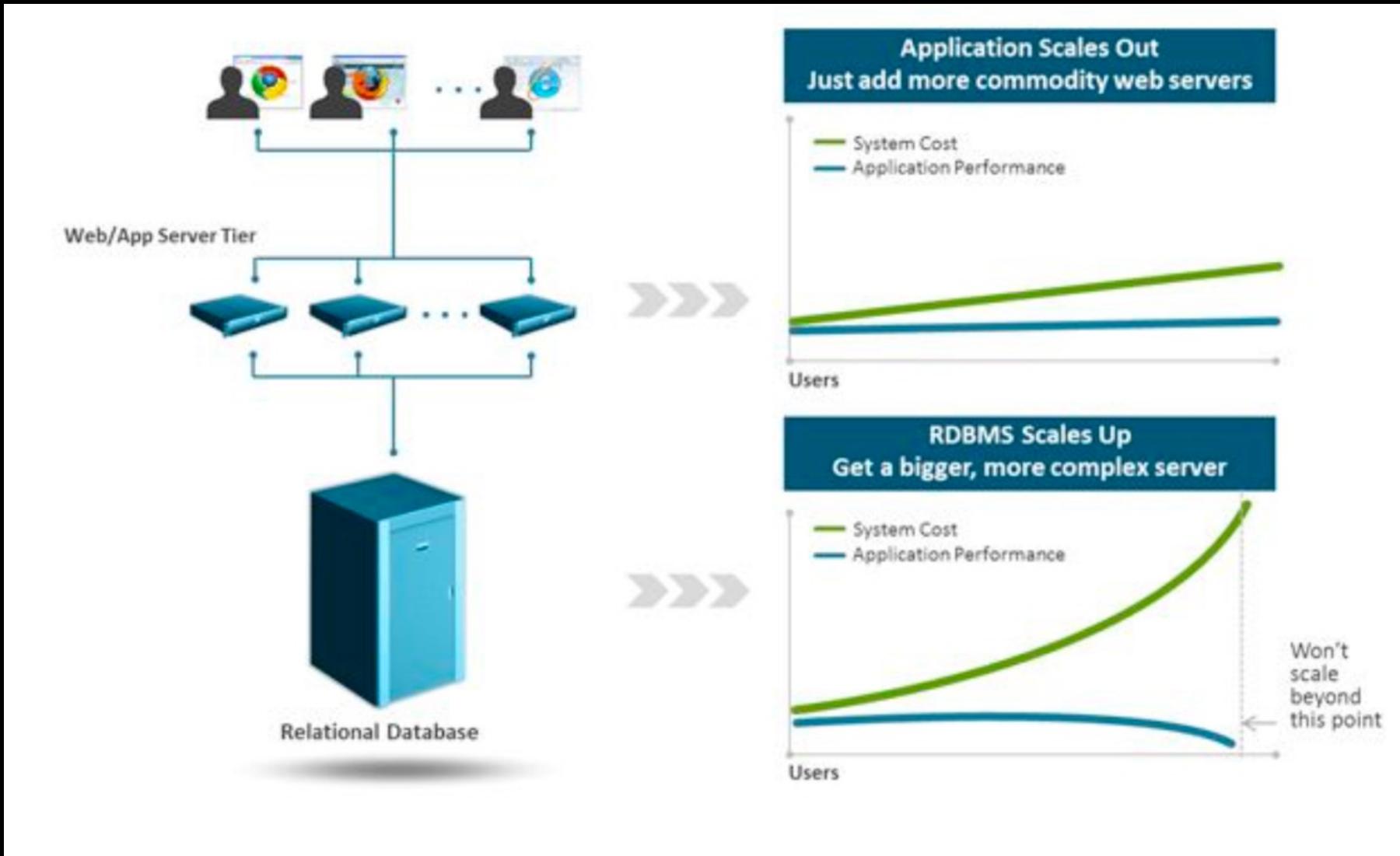
Lastly, the main user interface device at the time RDBMSs were architected was the dumb terminal, and vendors imagined operators inputting queries through an interactive terminal prompt. Now it is a powerful personal computer connected to the World Wide Web. Web sites that use OLTP DBMSs rarely run interactive transactions or present users with direct SQL

Let's get to it: Major Trends

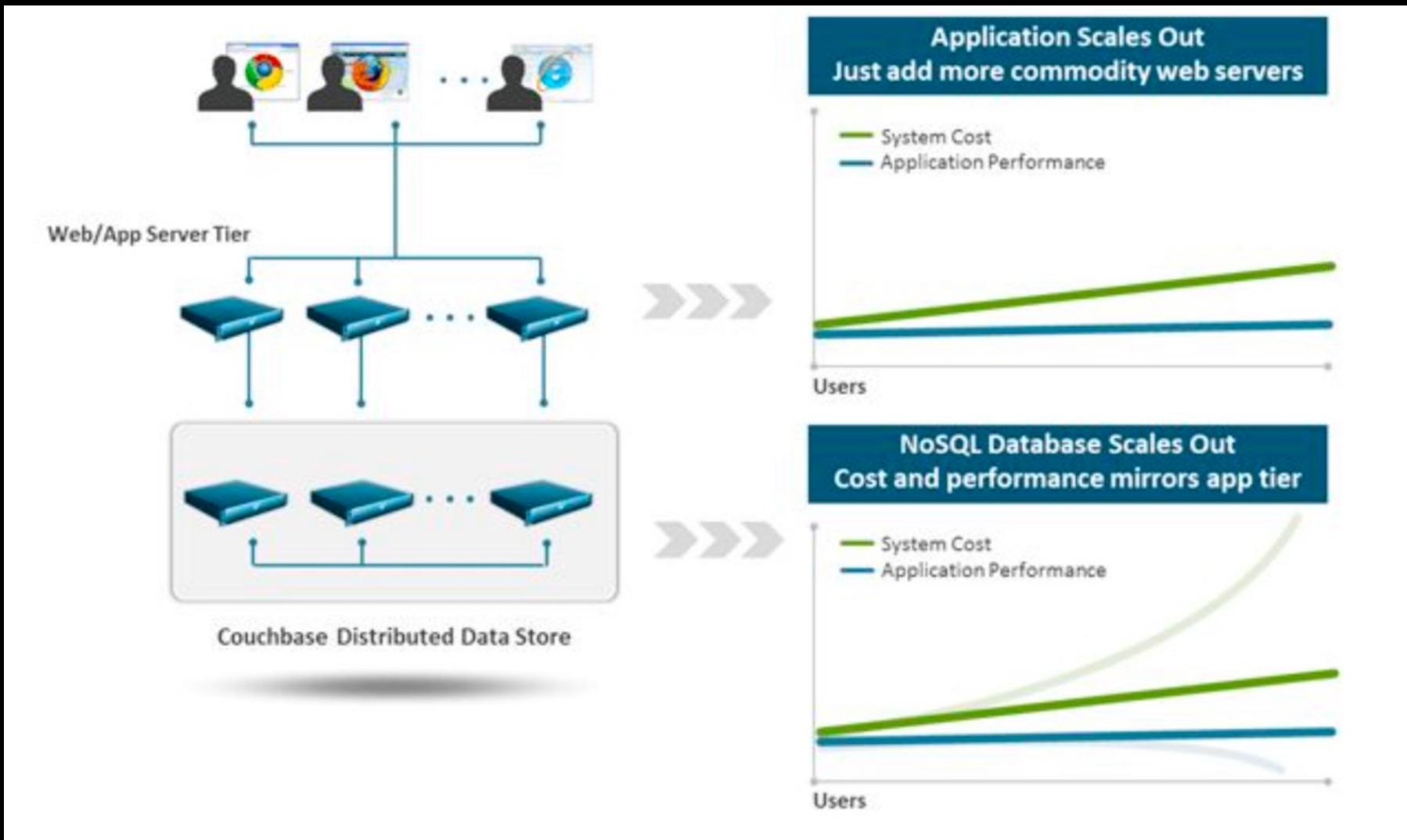
- **More hardware**
 - Better servers
 - More servers → Cloud
- **More data**
 - More quantity of data
 - More types of data
 - Still want fast systems
 - Less structure → less need for complexity



Scaling Up



Scaling Out





RDBMS

NoSQL

NoSQL

- **Data model**
 - Not relational
 - No formally described structure (schema-less)
- **Interface**
 - Not only SQL ( NoSQL name)
 - Proprietary, REST, CQL etc.
- **Architecture**
 - Usually **distributed**
- **Mostly not ACID compliant**
 - Consistency/Availability tradeoff (**CAP theorem**)
- **Mostly open source**

Key-Value Stores

- **Associative Array**

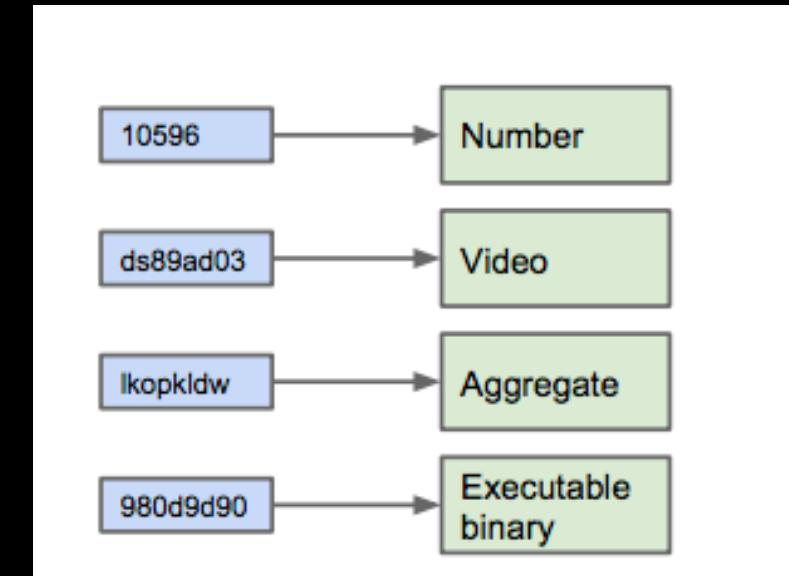
- Unique key points to a value
- Value contents unknown

- **Can not be queried**

- GET / PUT only
- Value can be an aggregate structure

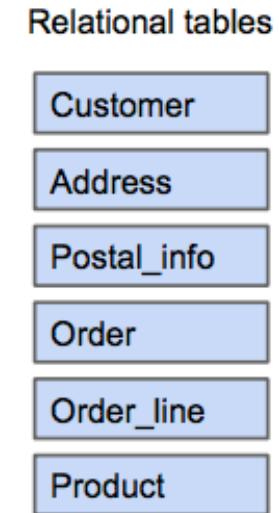
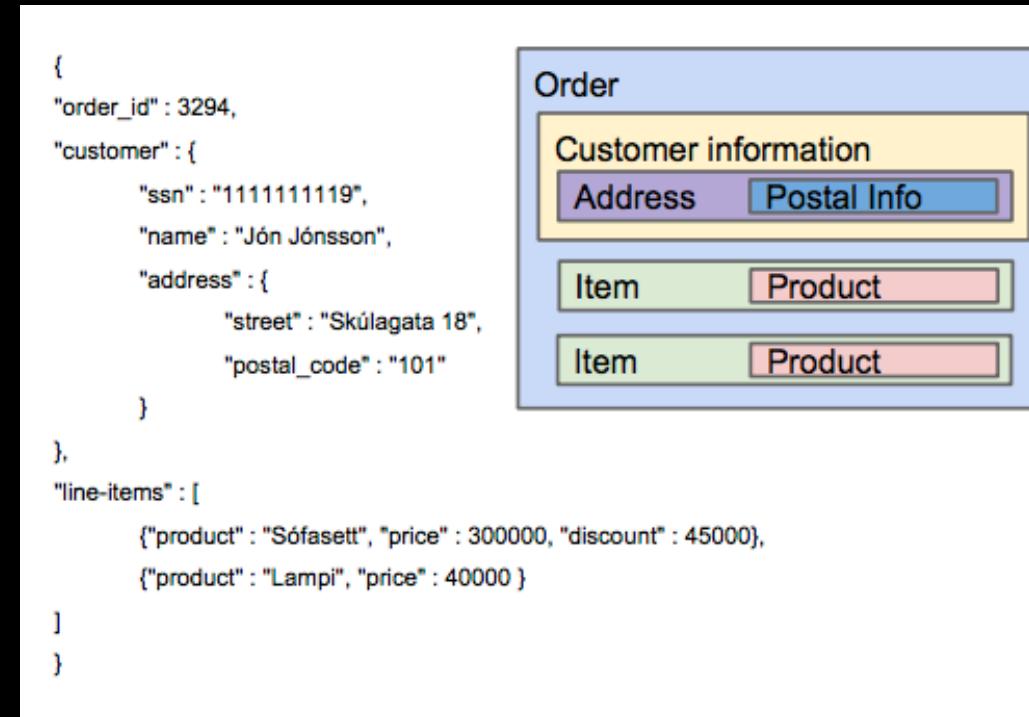
- **Examples**

- Riak, Voldemort, MemcacheDB, redis



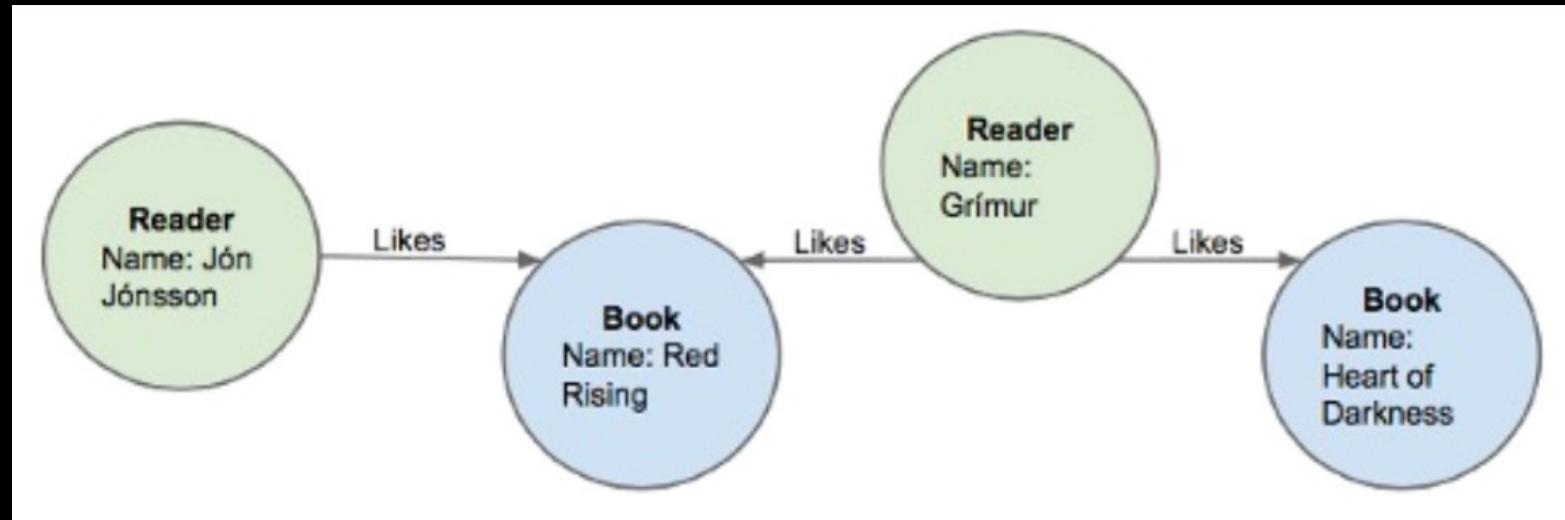
Document Stores

- Each value is a document
 - Most often JSON
 - Unique keys used for retrieval
- You can query into the document
 - More transparent than key-value stores
- The document is an aggregate structure
- Examples
 - Lotus Notes, CouchDB, MongoDB



Graph Stores

- Nodes = Entities
- Edges = Relationships, directional
- Properties = Entity descriptors
- Examples
 - neo4j, Allegro, InfiniGraph, OrientDB



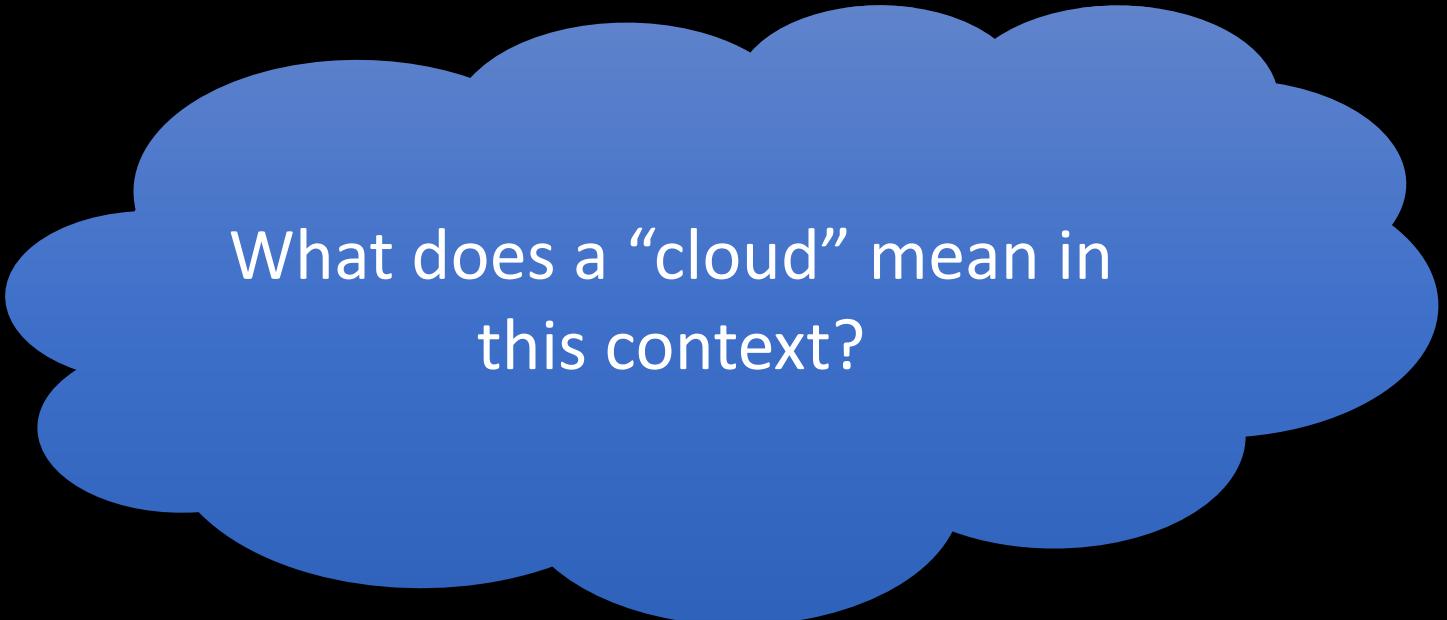
NoSQL

- **Data model**
 - Not relational
 - No formally described structure (schema-less)
- **Interface**
 - Not only SQL ( NoSQL name)
 - Proprietary, REST, CQL etc.
- **Architecture**
 - Usually distributed
- **Mostly not ACID compliant**
 - Consistency/Availability tradeoff ( CAP theorem)
- **Mostly open source**



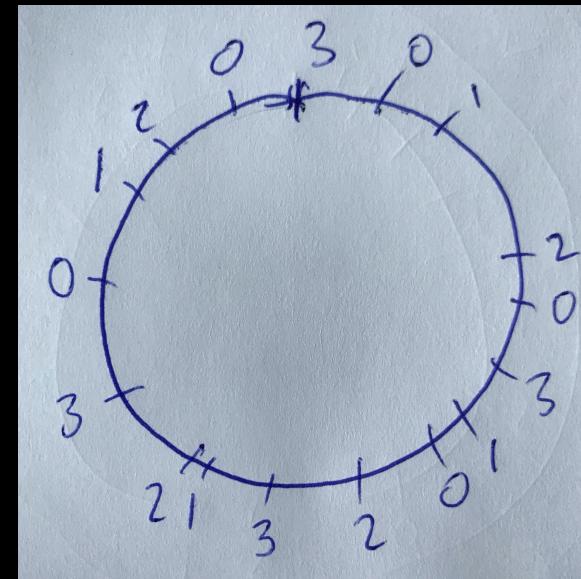
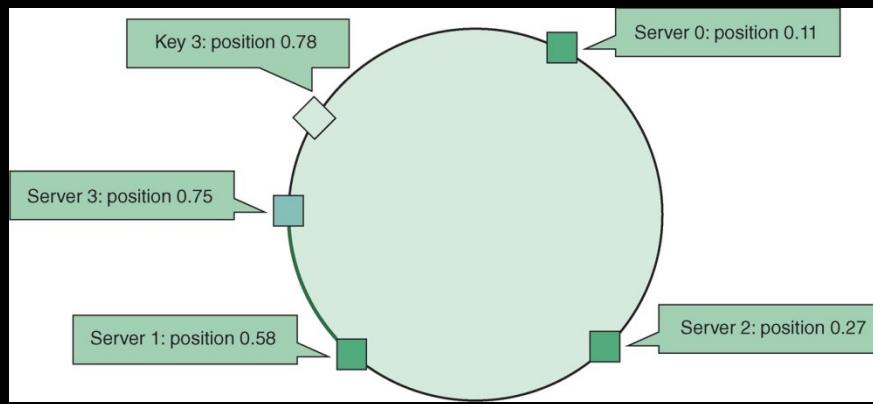
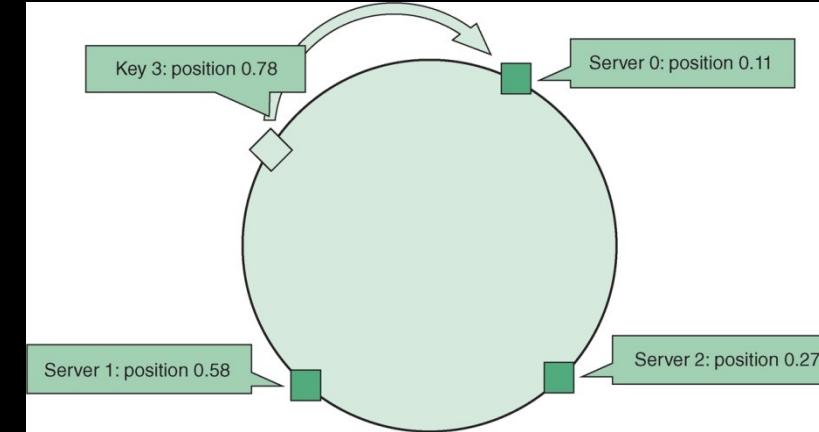
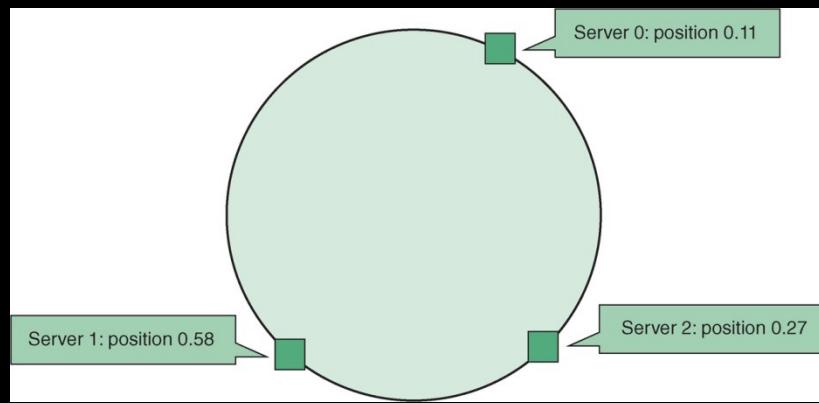
Goals of Distributed Storage

- Workload sharing – balance
- Redundancy (replicas) – failure handling

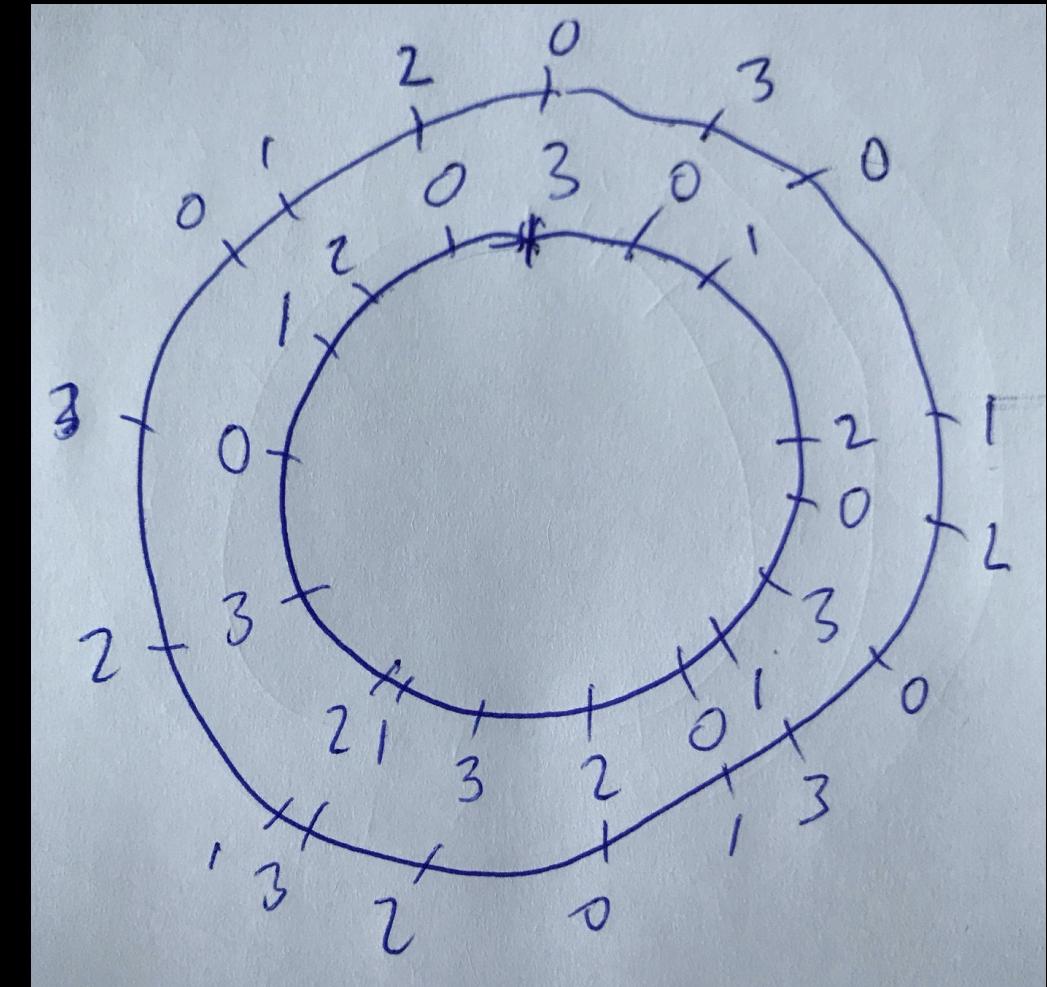
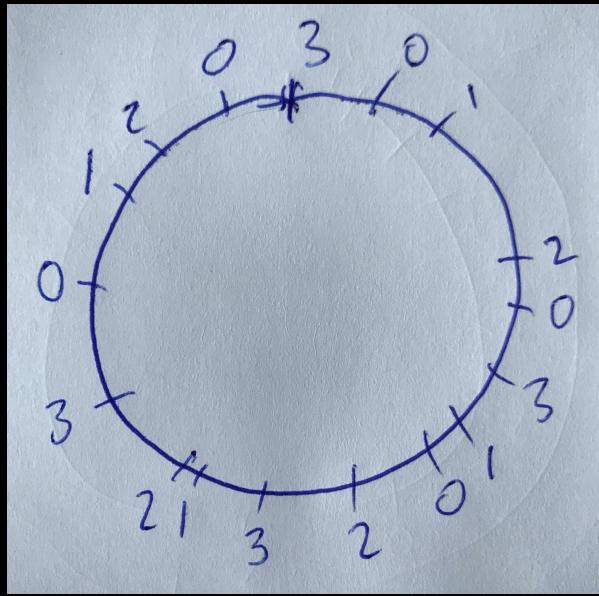


What does a “cloud” mean in
this context?

Consistent Hashing: Balance



Consistent Hashing: Redundancy



NoSQL

- **Data model**
 - Not relational
 - No formally described structure (schema-less)
- **Interface**
 - Not only SQL ( NoSQL name)
 - Proprietary, REST, CQL etc.
- **Architecture**
 - Usually **distributed**
- **Mostly not ACID compliant**
 - Consistency/Availability tradeoff (**CAP theorem**)
- **Mostly open source**



Replica Consistency

- **Sequential (or strong) consistency:** All updates are seen by all processes in the same order. As a result, the effects of an update are seen by all observers. There is no inconsistency.
 - Roughly the same as Isolation in ACID
- **Weak consistency:** Observers might see inconsistencies among replicas
- **Eventual consistency:** A form of weak consistency, where at some point, in case there is no failure, all replicas will reflect the last update.

Tunable Consistency

- Not a binary decision
 - N replicas, R reads, W writes
- $R = W = 1$ gives eventual consistency
- $R + W > N$ gives strong consistency

Example:

- if the replication factor is 3, then the consistency level of the reads and writes combined must be at least 4. Read operations using 2 out of 3 replicas to verify the value and write operations using 2 out of 3 replicas to verify the value will result in strong consistency.

CAP Theorem

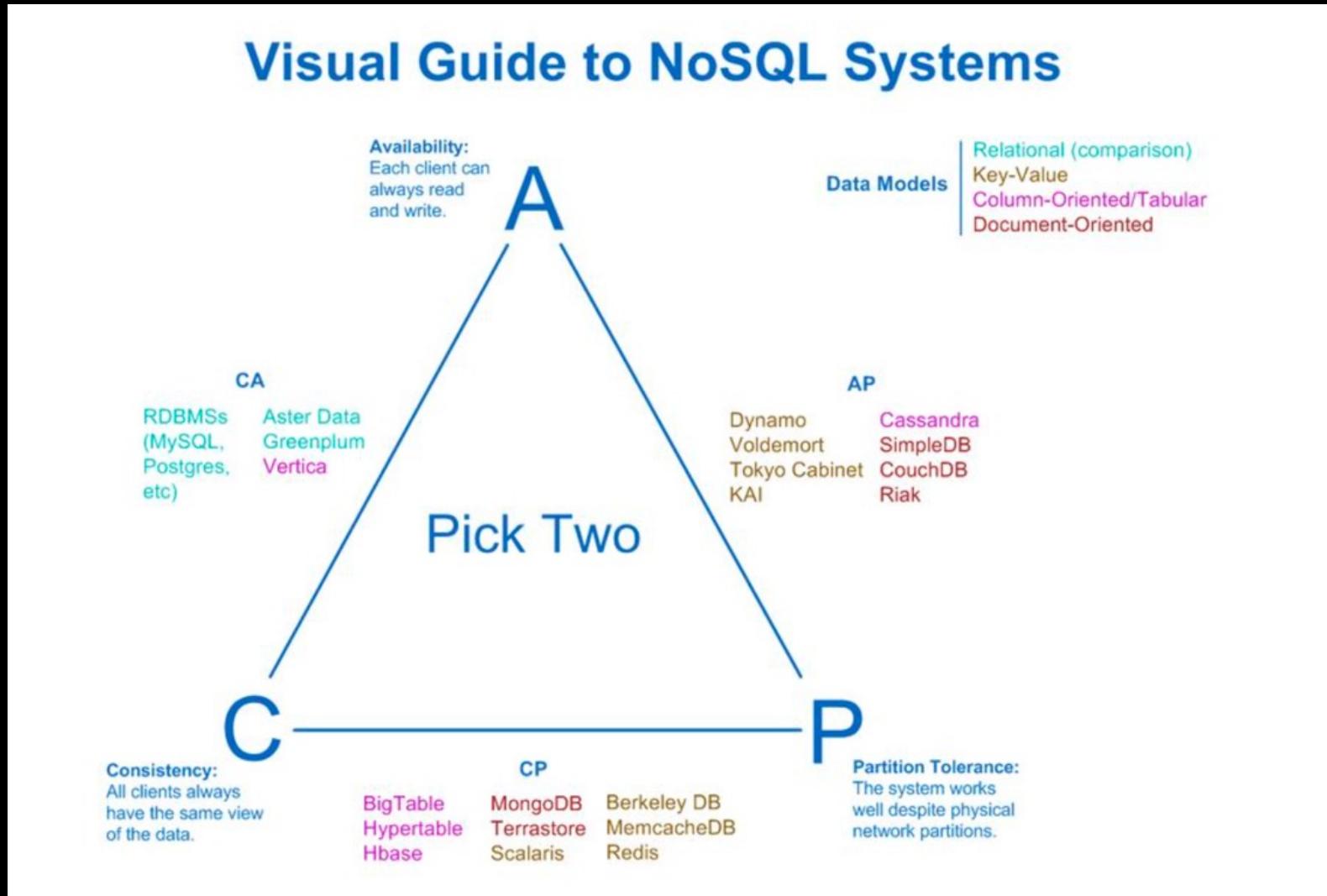
- **C = Consistency**
 - Readers read most recent update
- **A = Availability**
 - An answer is returned
- **P = Partitions**
 - The network becomes disconnected

Incorrect (but Typical) Formulation

- You can only get two of
 - Consistency
 - Availability
 - Partition Tolerance

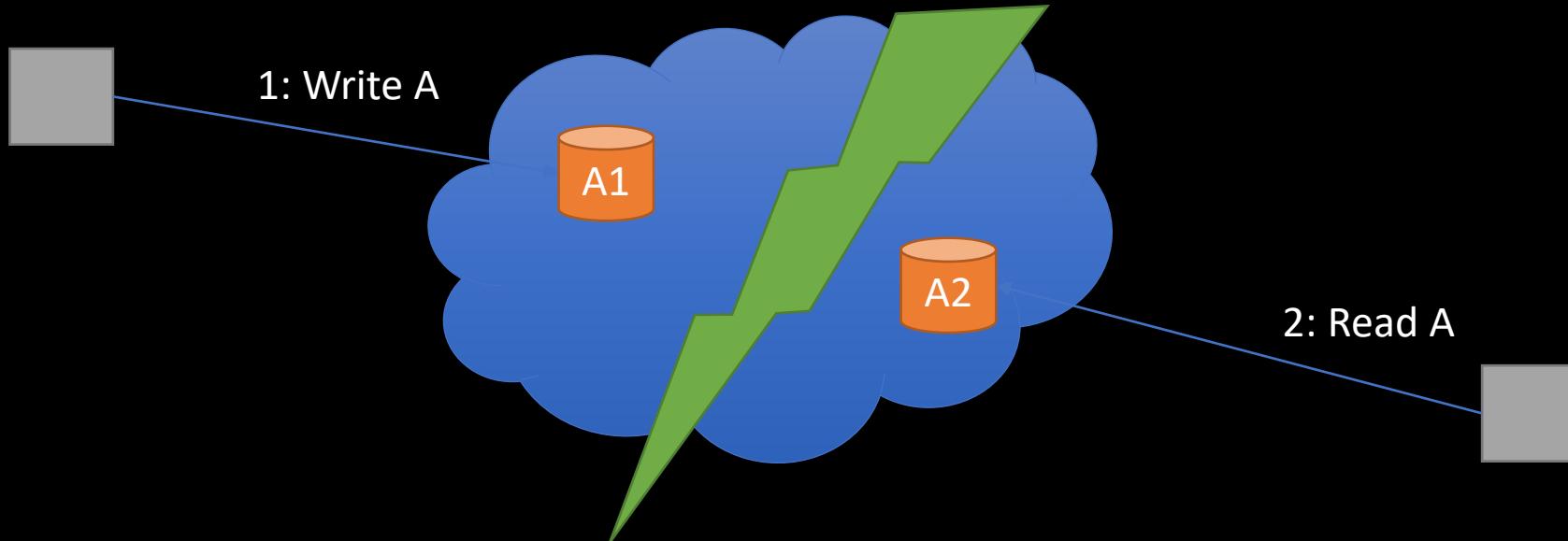


Resulting Classification



Correct (but Useless) Formulation

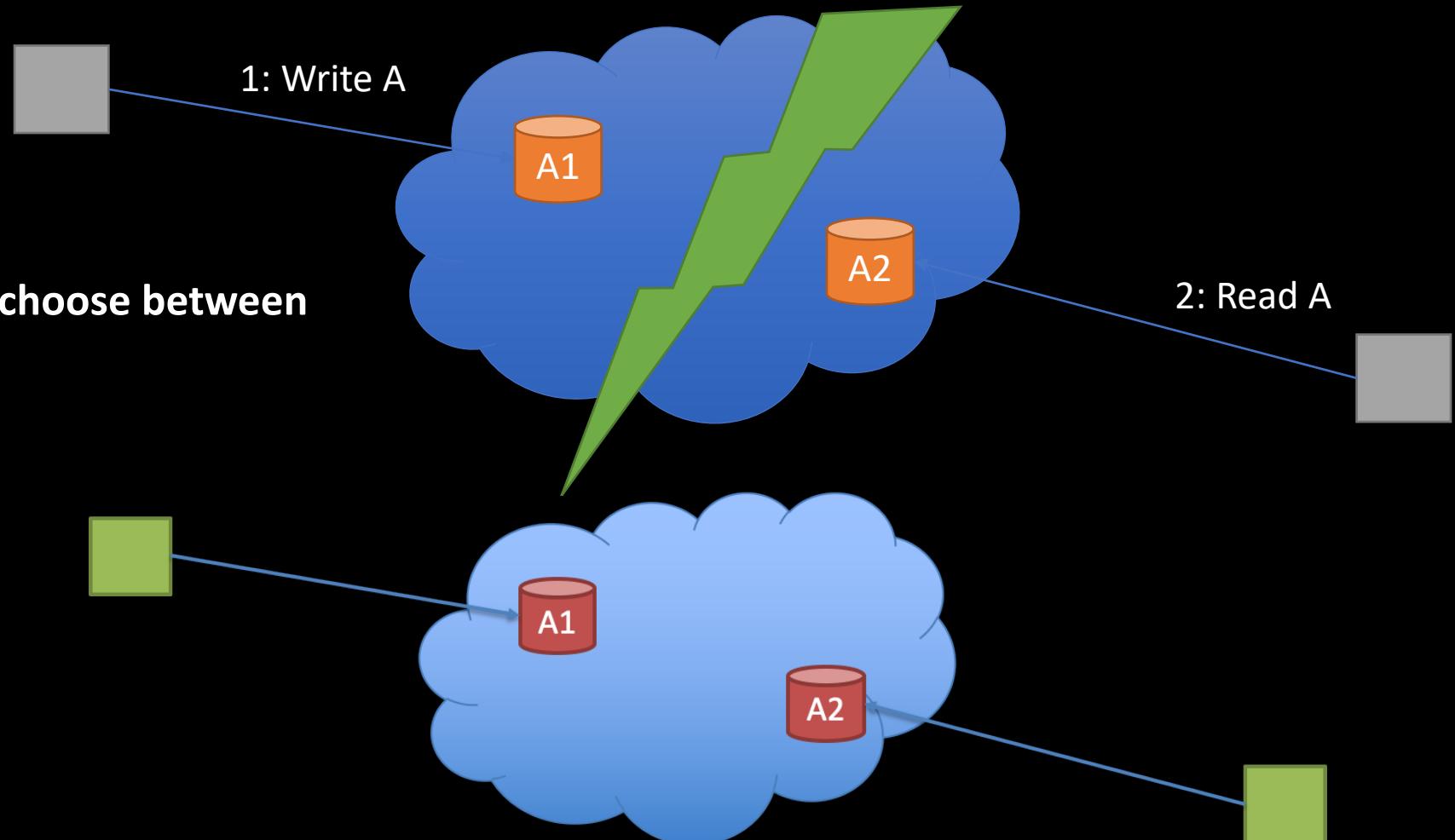
- In a partitioned network, choose between
 - Consistency
 - Availability



PACELC (*useful!*) Formulation

- In a partitioned network, choose between

- Consistency
- Availability



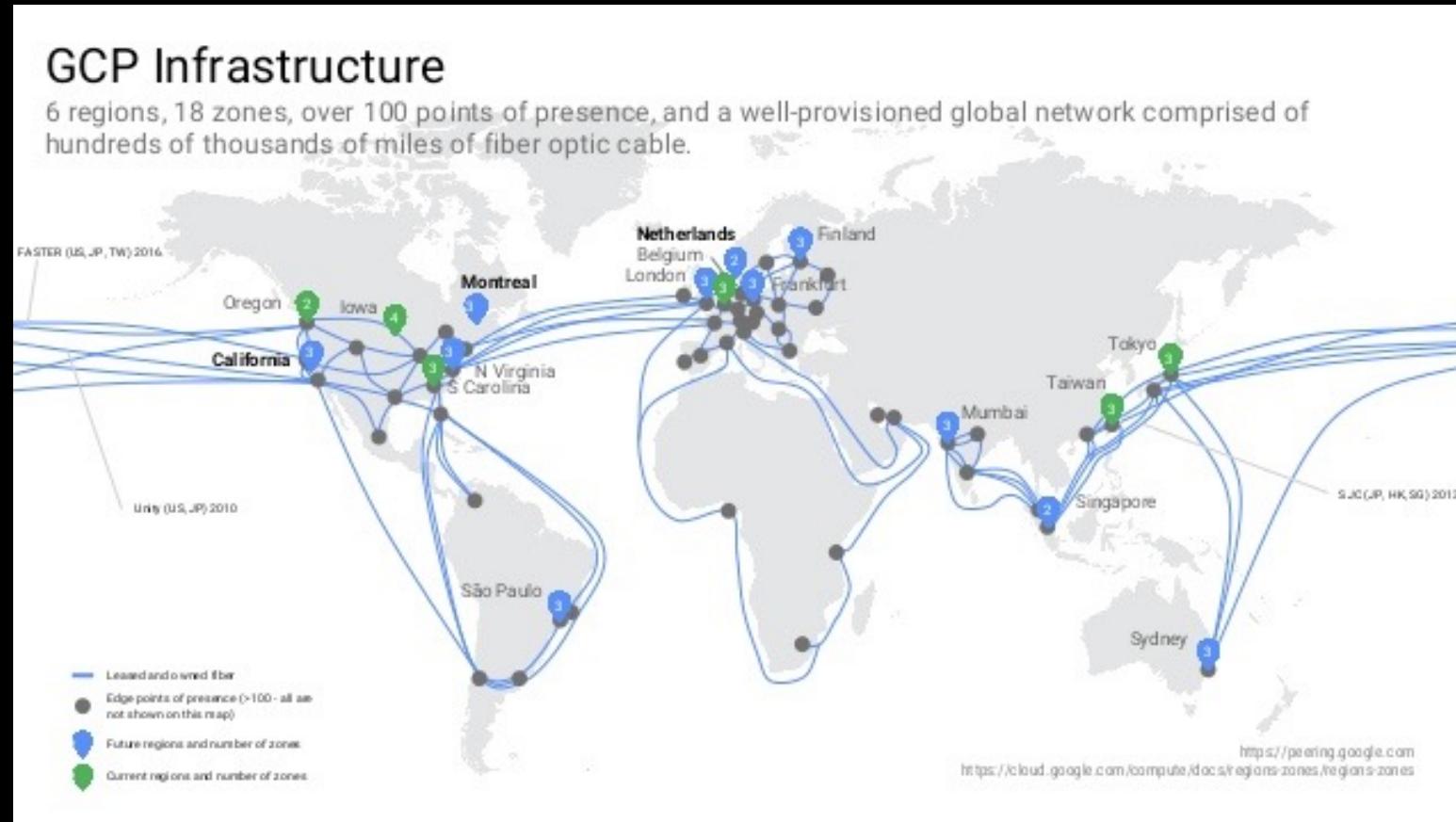
2012: Google's Spanner

Cloud Spanner: The best of the relational and non-relational worlds

	CLOUD SPANNER	TRADITIONAL RELATIONAL	TRADITIONAL NON-RELATIONAL
Schema	✓ Yes	✓ Yes	✗ No
SQL	✓ Yes	✓ Yes	✗ No
Consistency	✓ Strong	✓ Strong	✗ Eventual
Availability	✓ High	✗ Failover	✓ High
Scalability	✓ Horizontal	✗ Vertical	✓ Horizontal
Replication	✓ Automatic	⟳ Configurable	⟳ Configurable

- Does it break the CAP theorem?

What is High Availability?



- “Cloud Spanner [...] serves data with low latency while maintaining transactional consistency and industry-leading 99.999% (five 9s) availability - 10x less downtime than four nines (<5 minutes per year).”

<https://cloud.google.com/spanner/>

2012: Google's Spanner

Cloud Spanner: The best of the relational and non-relational worlds

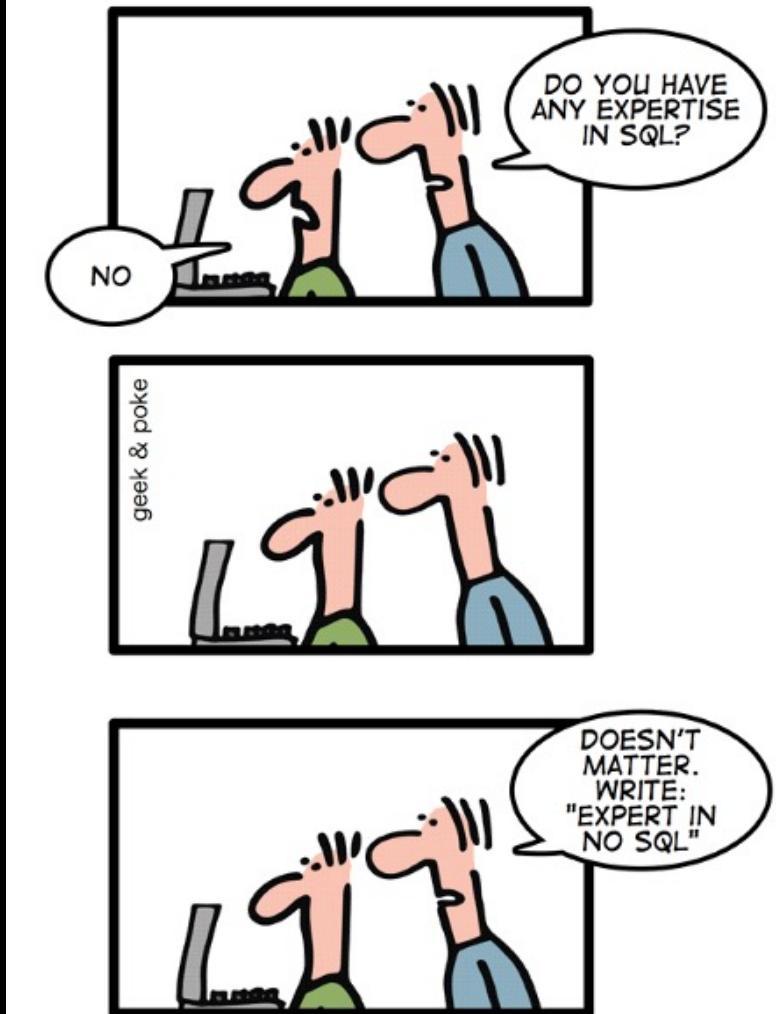
	CLOUD SPANNER	TRADITIONAL RELATIONAL	TRADITIONAL NON-RELATIONAL
Schema	✓ Yes	✓ Yes	✗ No
SQL	✓ Yes	✓ Yes	✗ No
Consistency	✓ Strong	✓ Strong	✗ Eventual
Availability	✓ High	✗ Failover	✓ High
Scalability	✓ Horizontal	✗ Vertical	✓ Horizontal
Replication	✓ Automatic	⟳ Configurable	⟳ Configurable

- Does it break the CAP theorem? **No!**

NoSQL

- A result of two trends:
 - More hardware
 - More data
- Data model
 - Not relational
 - No formally described structure (schema-less)
- Architecture
 - Usually **distributed**
- Mostly not ACID compliant
 - CAP = Consistency/Availability tradeoff

HOW TO WRITE A CV





Takeaways

- **Main-Memory RDBMS:**
 - No buffer management
 - Better cache utilization
 - Light-weight logging
 - No concurrency control
- **Scale Out:**
 - Better than scale up
- **CAP Theorem:**
 - Consistency
 - Availability
 - Partitions
 - Pick between availability and consistency in case of network partition



What is next?

- **Next Lecture:**
 - Big Data

Introduction to Database Systems

I2DBS – Spring 2023

- Week 12:
- Big Data Processing
- Cross-Platform Data Processing

Jorge-Arnulfo Quiané-Ruiz

Some of the slides courtesy of
Zoi Kaoudi

Information

- **Homework 4:**

- Deadline: May 14th, 2023

- **Trial Exam**

- I will be open on May 8th and will close on May 12th , 2023
 - Once started, you have 4 hours to finish it

- **Final Exam**

- It will take place in the following rooms:



Profile of the Week

Jeff Dean

A Big Data Systems Pioneer

- 1968: Born in Colorado, USA
- 1996: PhD in CS from the University of Washington
- 1999: Google
- **Co-Inventor of:** GFS, MapReduce, TensorFlow, ...
- **Founder of:** Google Brain team



Hector Garcia Molina

A Database & Distributed Systems Leader

- 1954: Born in Mexico City, Mexico
- 1981: PhD in CS from Stanford University
- 1981: Associate Professor at Princeton University
- 1991: Professor at Stanford University
- 1999: ACM SIGMOD Contributions Award
- 1994: Leader of the Stanford Digital Library Project
 - Google spun off from this project!
- 2004: ACM SIGMOD Contributions Award
- 2019: R.I.P.



Big Data

From distributed systems to big data systems

- ♦ Problems with distributed systems
 - ♦ Nondeterministic behaviour
 - ♦ Partial failures
 - ♦ Debugging becomes very hard
 - ♦ High probability of large variance in runtime/throughput performance

From distributed systems to big data systems

- ♦ Problems with distributed systems
 - ♦ Nondeterministic behaviour
 - ♦ Partial failures
 - ♦ Debugging becomes very hard
 - ♦ High probability of large variance in runtime/throughput performance



Need for new tools

Need for new tools

- ◆ Distributed File Systems (DFS)
 - ◆ Store petabytes of data in a cluster
 - ◆ Transparently handle fault-tolerance and replication
- ◆ Parallel Processing Platforms
 - ◆ Offer a programming model to allow developers to easily write distributed applications
 - ◆ Alleviate developer from handling concurrency, network communication, and machine failures
 - ◆ Move computation to data, not data to computation

Google cluster idea (2003)

- ◆ **Failures are the norm**
 - ◆ 1 server → may stay up three years (1,000 days)
 - ◆ 10,000 servers → expect to lose 10 per day
 - ◆ Ultra-reliable hardware doesn't really help (at large scales, even most reliable hardware fails, albeit less often)
 - ◆ Software still needs to be fault-tolerant
- ◆ **Data is growing:** either large files or billions of small files
- ◆ **Append-only** instead of overwriting
 - ◆ Random writes are practically nonexistent

Google cluster idea (2003)

- ◆ **Failures are the norm**
 - ◆ 1 server → may stay up three years (1,000 days)
 - ◆ 10,000 servers → expect to lose 10 per day
 - ◆ Ultra-reliable hardware doesn't really help (at large scales, even most reliable hardware fails, albeit less often)
 - ◆ Software still needs to be fault-tolerant
- ◆ **Data is growing:** either large files or billions of small files
- ◆ **Append-only** instead of overwriting
 - ◆ Random writes are practically nonexistent

Use of commodity machines!



Google cluster idea (2003)

- ◆ **Failures are the norm**
 - ◆ 1 server → may stay up three years (1,000 days)
 - ◆ 10,000 servers → expect to lose 10 per day
 - ◆ Ultra-reliable hardware doesn't really help (at large scales, even most reliable hardware fails, albeit less often)
 - ◆ Software still needs to be fault-tolerant
- ◆ **Data is growing:** either large files or
- ◆ **Append-only** instead of overwriting
 - ◆ Random writes are practically nonexistent

Use of commodity machines!

Goal: throughput/\$ and not peak performance

Today's lecture

- ◆ Distributed file systems (HDFS)
- ◆ Distributed data processing paradigms (MapReduce)
- ◆ Distributed data processing platforms (Hadoop, Spark)



Today's lecture

- ♦ **Distributed file systems (DFS)**
- ♦ Distributed data processing paradigms (MapReduce)
- ♦ Distributed data processing platforms (Hadoop, Spark)



Google File System (GFS)

- ◆ Stores very large data files **efficiently** and **reliably**
- ◆ Files divided into **fixed-sized chunks** (eg., 64MB)
 - ◆ Stored as Linux files

Google File System (GFS)

Google File System (GFS)

- ◆ Multiple **chunkservers**
 - ◆ Store chunks on local disks
 - ◆ Each chunk replicated to multiple chunkservers
 - ◆ No caching of remote chunks (not worth it)

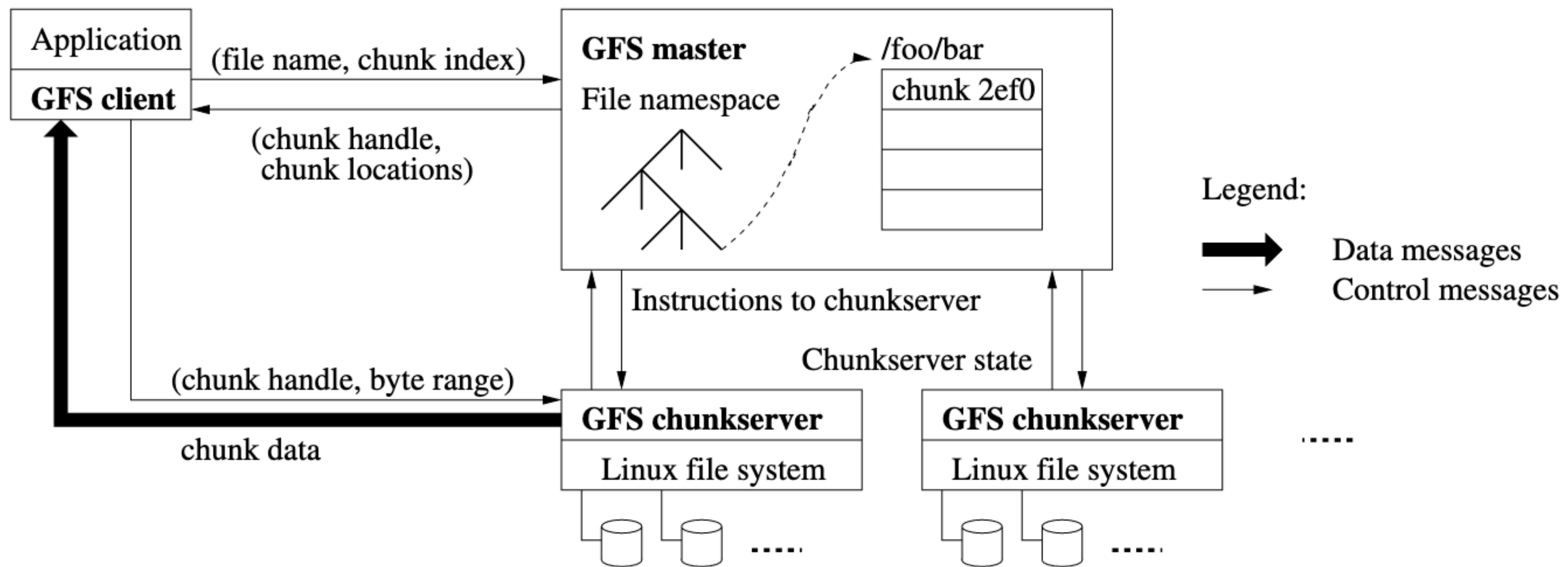
Google File System (GFS)

- ◆ One **master**
 - ◆ Each chunk gets a chunk handle from the master
 - ◆ Maintains all file system metadata
 - ◆ Talks to chunkservers periodically
- ◆ Multiple **chunkservers**
 - ◆ Store chunks on local disks
 - ◆ Each chunk replicated to multiple chunkservers
 - ◆ No caching of remote chunks (not worth it)

Google File System (GFS)

- ◆ One **master**
 - ◆ Each chunk gets a chunk handle from the master
 - ◆ Maintains all file system metadata
 - ◆ Talks to chunkservers periodically
- ◆ Multiple **chunkservers**
 - ◆ Store chunks on local disks
 - ◆ Each chunk replicated to multiple chunkservers
 - ◆ No caching of remote chunks (not worth it)
- ◆ Multiple **clients**
 - ◆ Talk to master for metadata operations
 - ◆ Metadata can be cached at clients
 - ◆ Read / write of data from chunkservers

GFS architecture



- ◆ Single master, multiple chunk servers
- ◆ Master is potential **single point of failure / scalability bottleneck**
 - ◆ Use shadow masters
 - ◆ Minimize master involvements (large chunks, only metadata)

Hadoop Distributed File System (HDFS)

- ♦ Distributed file systems manage the storage across a network of machines
- ♦ HDFS is Hadoop's flagship filesystem

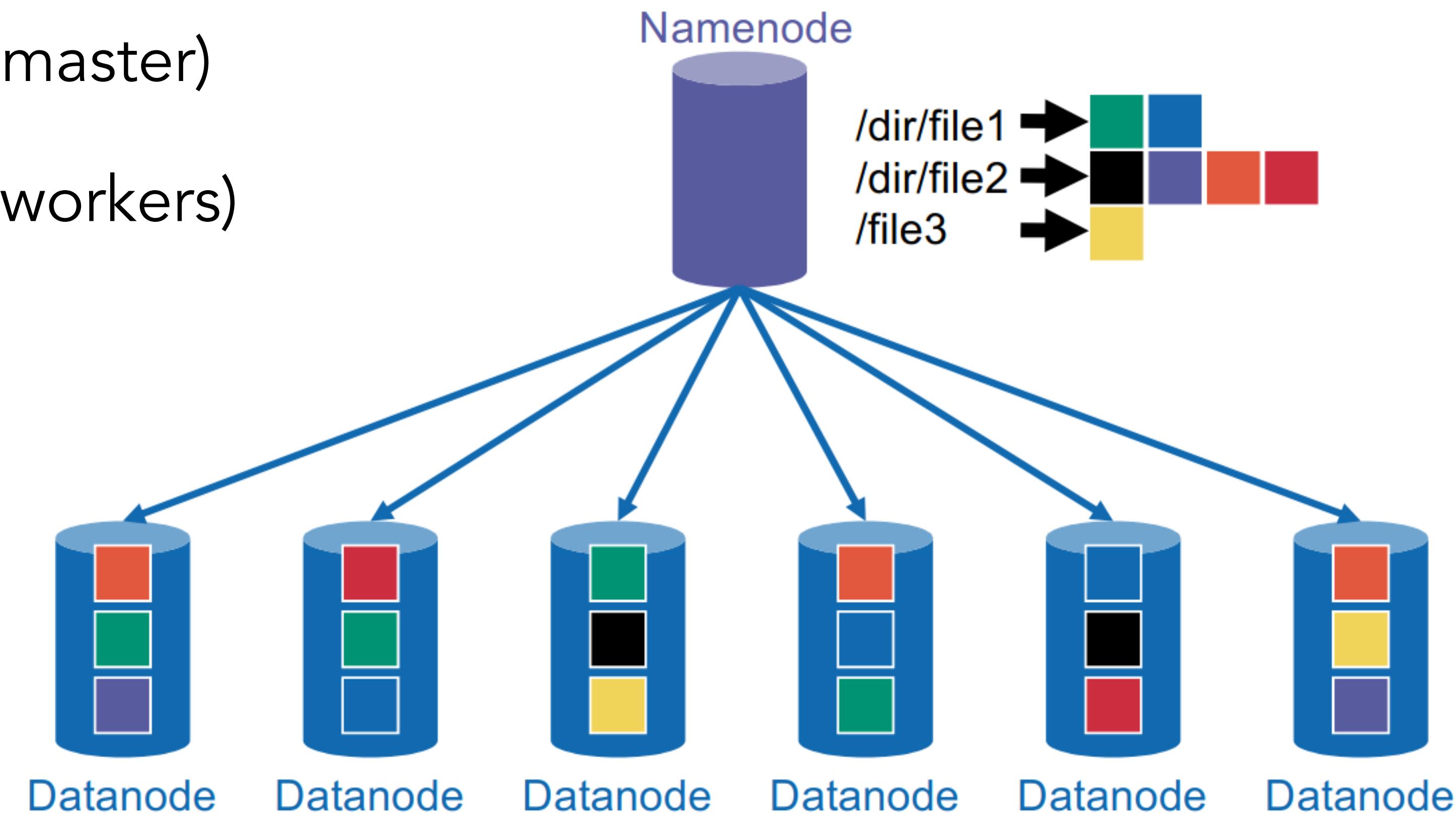
HDFS Block

- ◆ HDFS stores very large files
- ◆ Files are **split** into block-sized chunks (by default 64MB)
- ◆ With this block abstraction:
 - ◆ A file can be larger than a single disk in the network
 - ◆ HDFS blocks are larger than disk blocks
 - ◆ Storage system is simplified (metadata for blocks)
 - ◆ **Replication** for fault-tolerance and availability is facilitated

HDFS architecture

- ◆ Two types of HDFS nodes

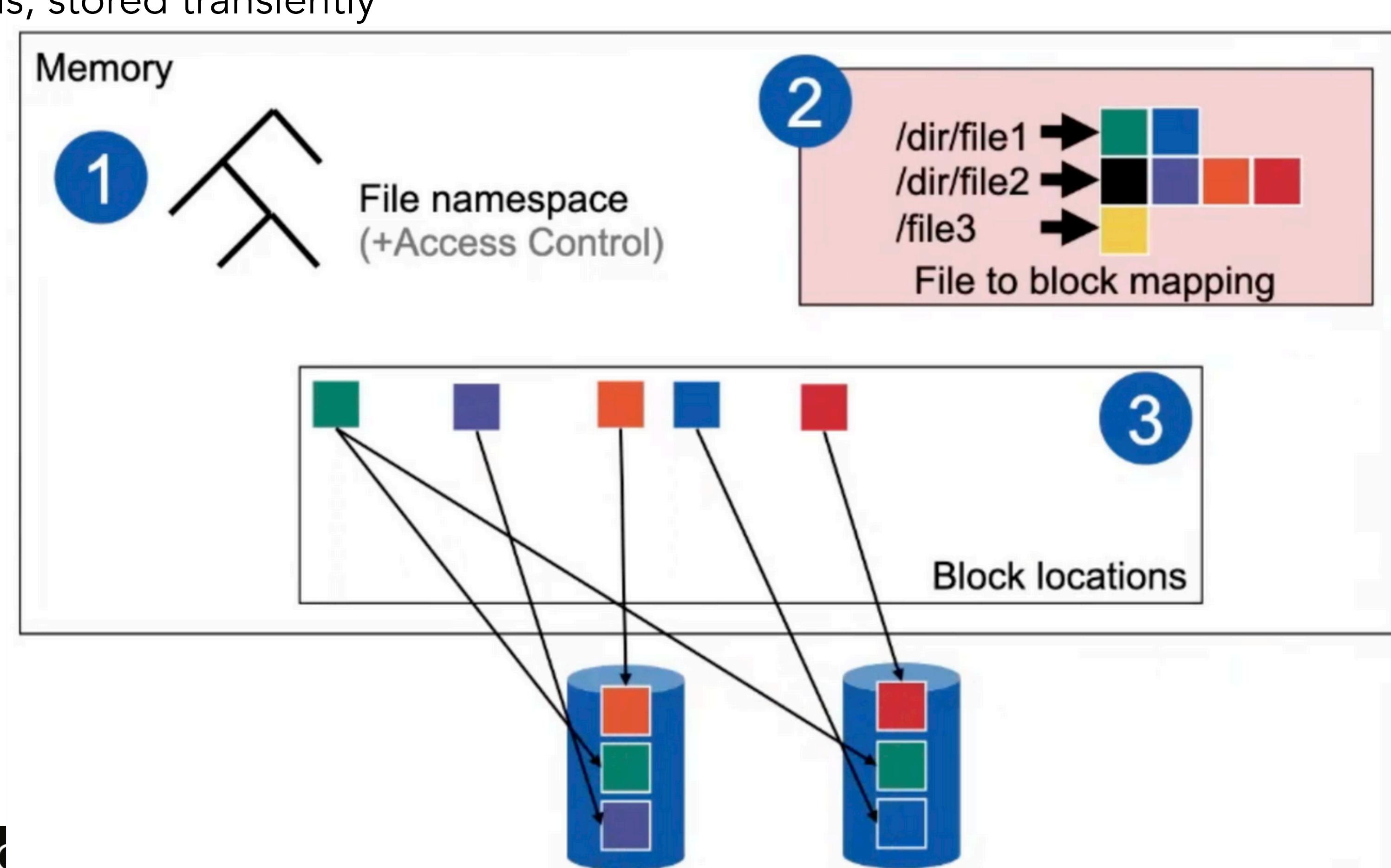
- ◆ One **namenode** (the master)
- ◆ Multiple **datanodes** (workers)



Namenode

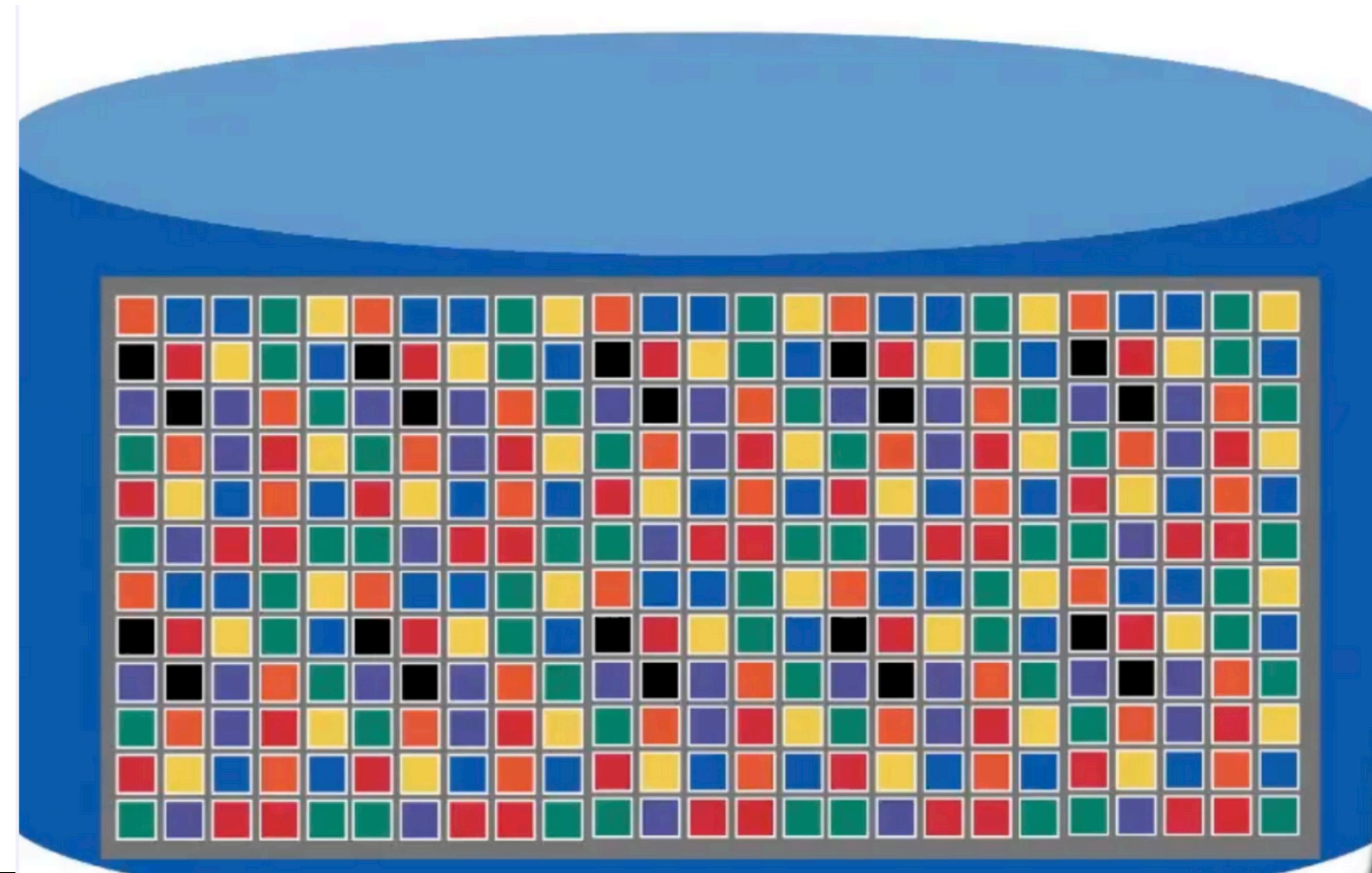
- ◆ Namenode manages the filesystem namespace

- ◆ File system tree and metadata, stored persistently
- ◆ Block locations, stored transiently



Datanode

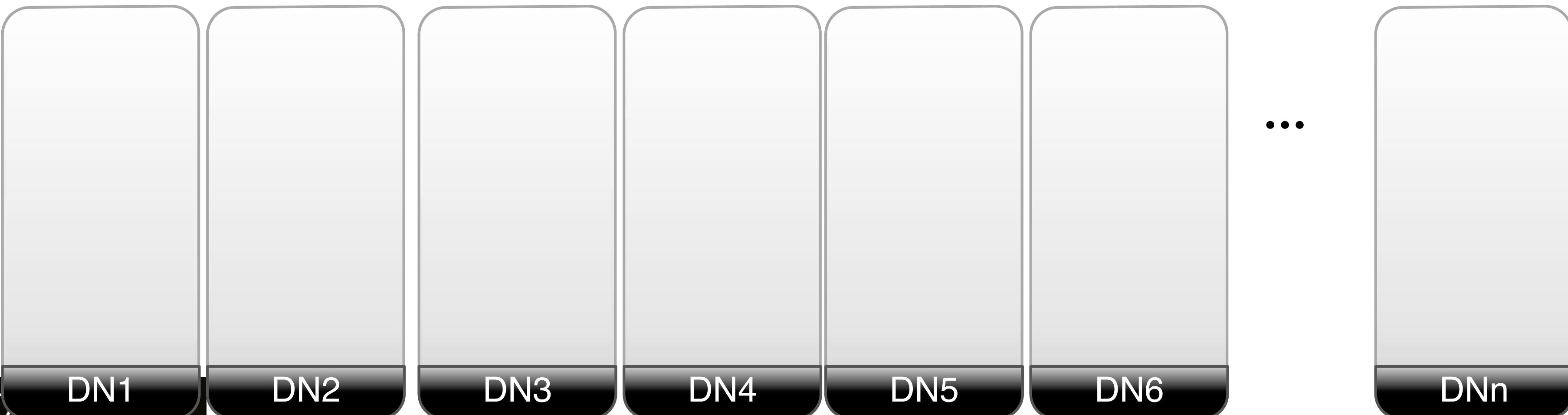
- ◆ Datanodes store and retrieve data blocks
 - ◆ When requested from clients or namenode
 - ◆ Also send list of stored blocks periodically to namenode



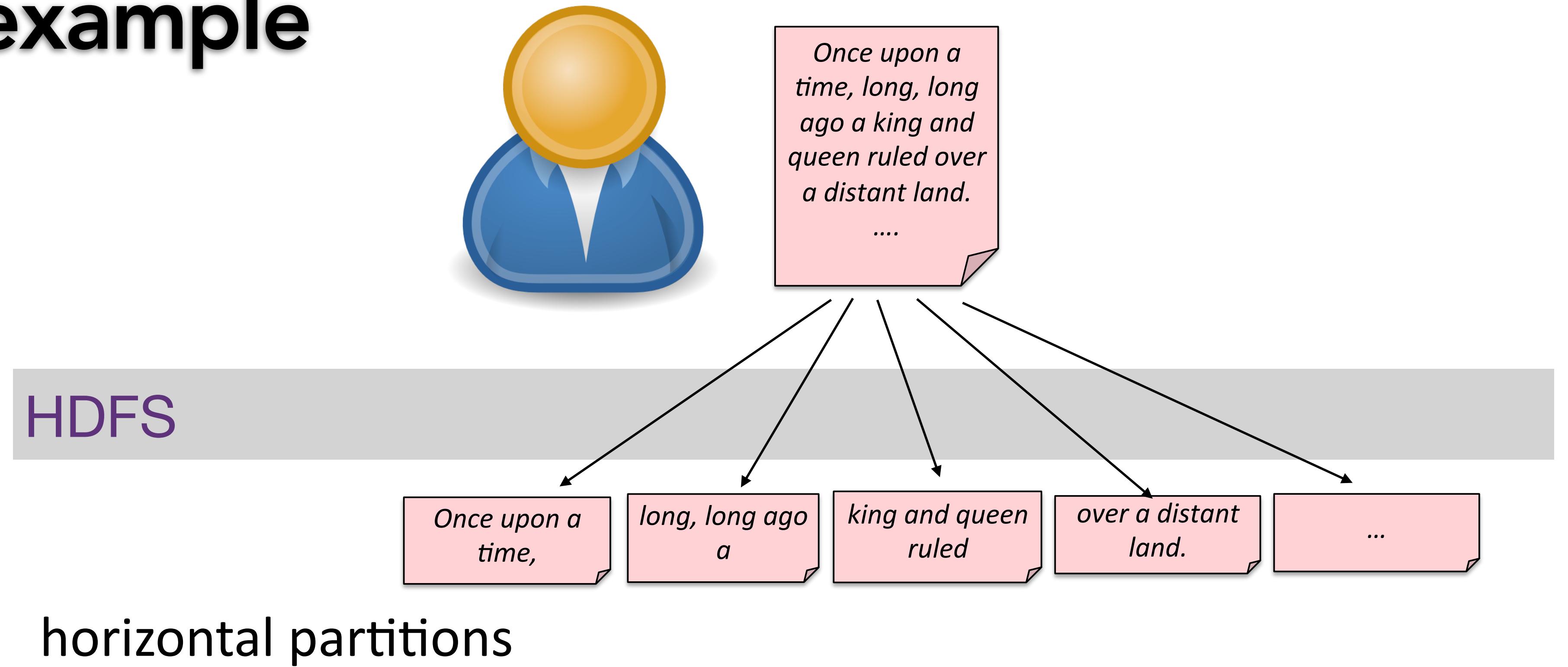
HDFS example



HDFS



HDFS example



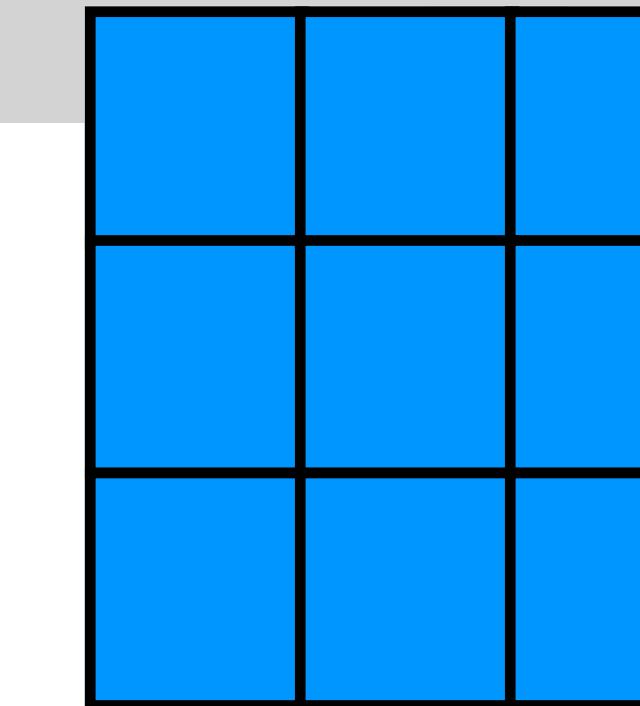
HDFS example



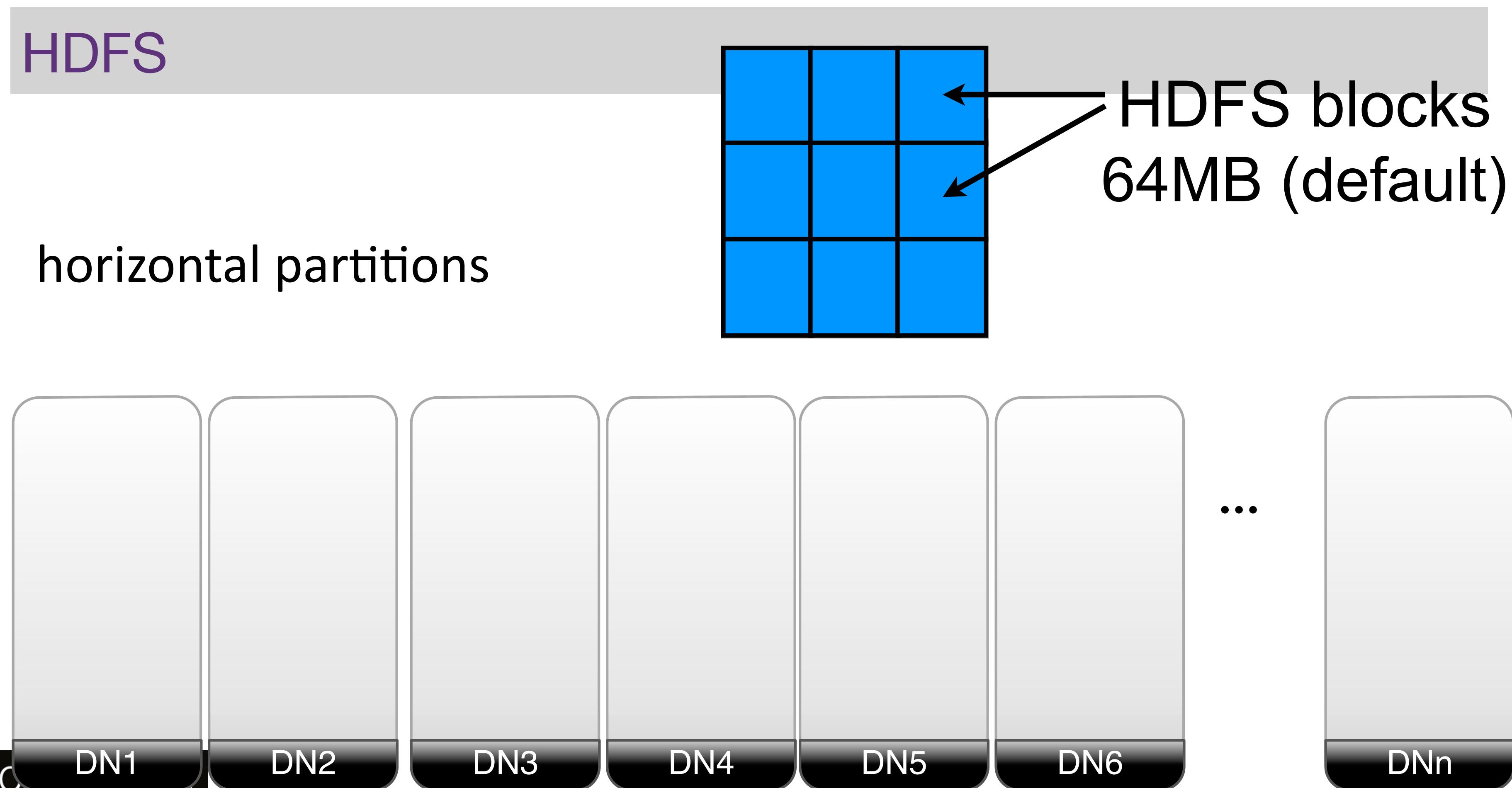
*Once upon a
time, long, long
ago a king and
queen ruled over
a distant land.
....*

HDFS

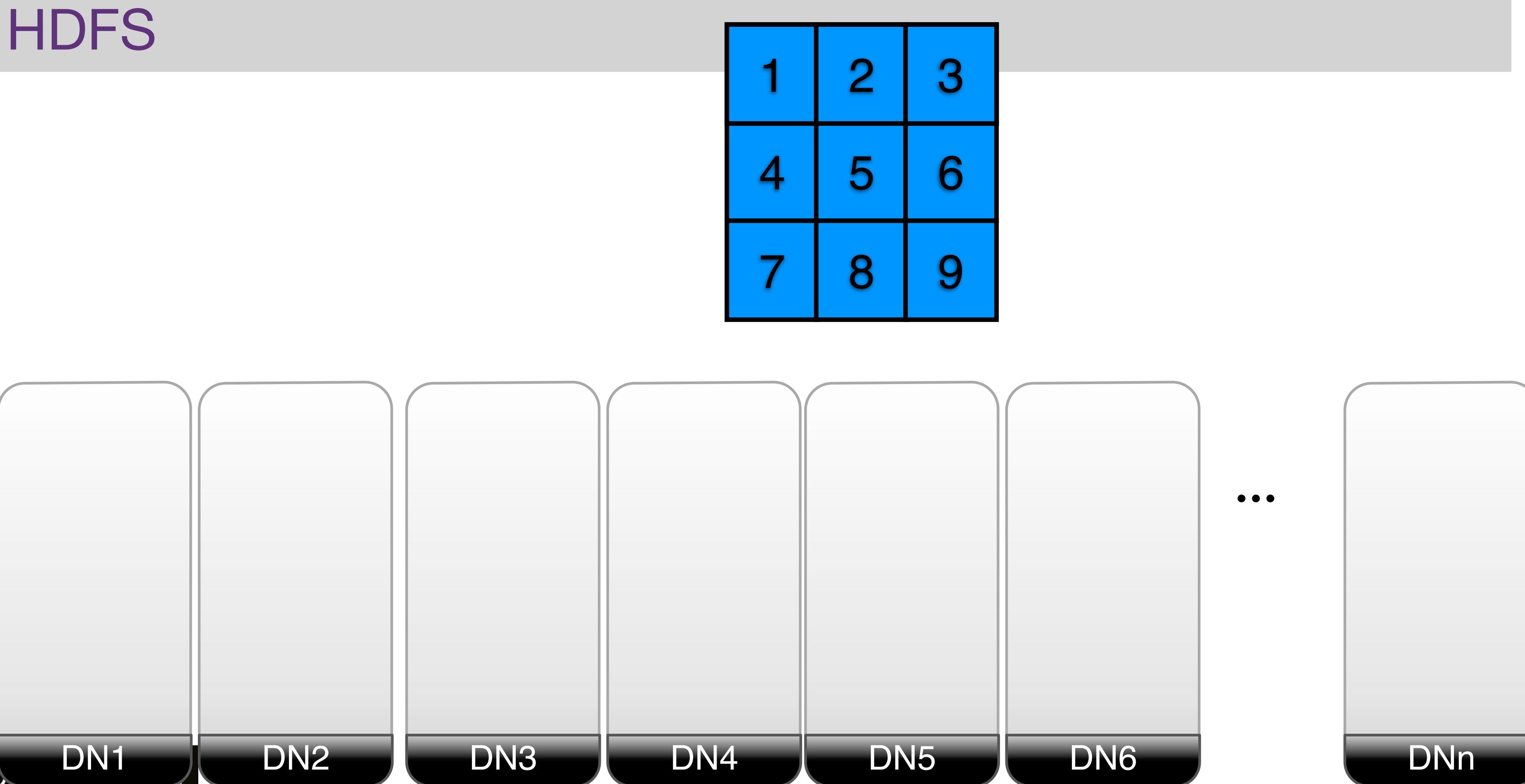
horizontal partitions



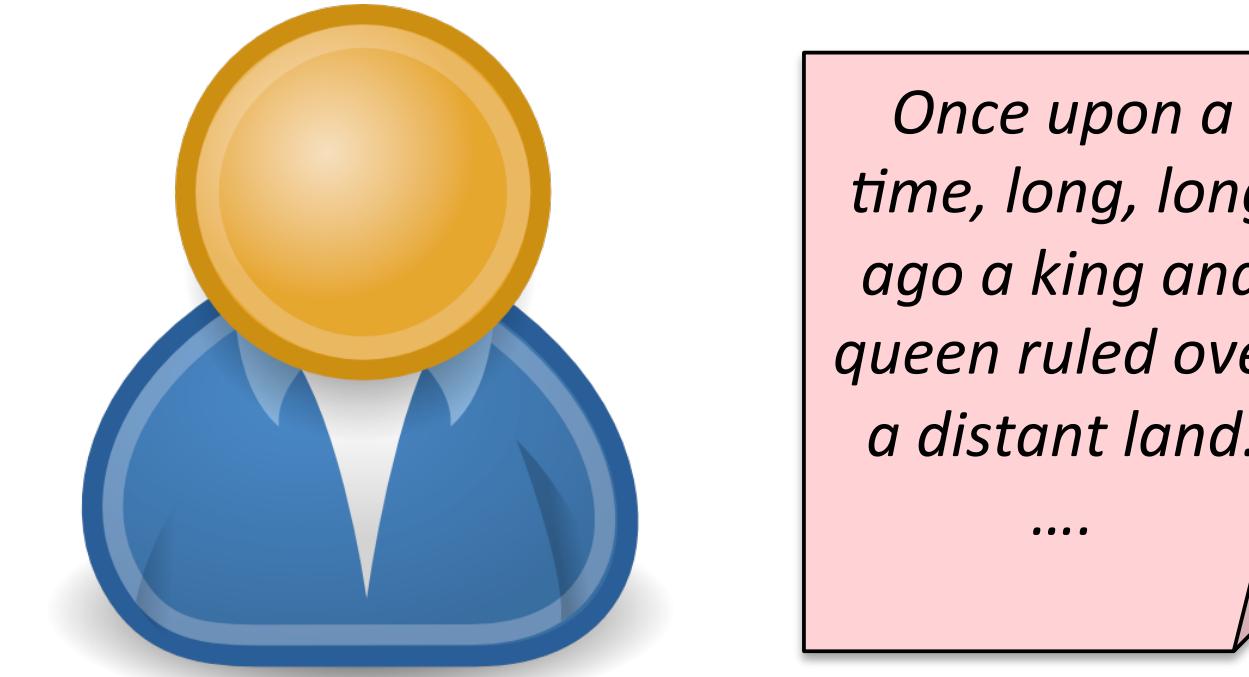
HDFS example



HDFS example



HDFS example

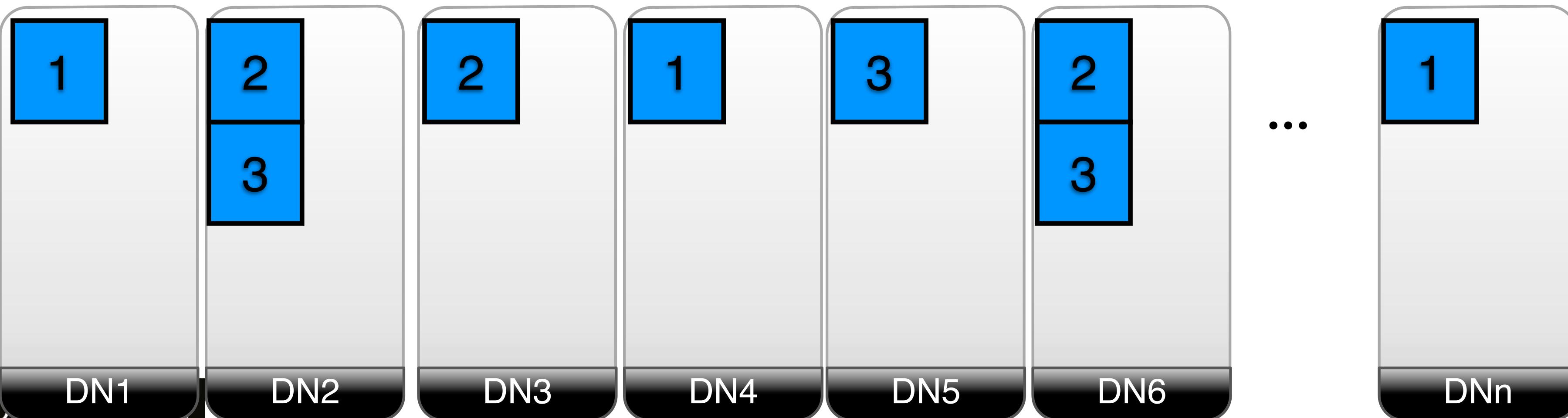


Once upon a time, long, long ago a king and queen ruled over a distant land.

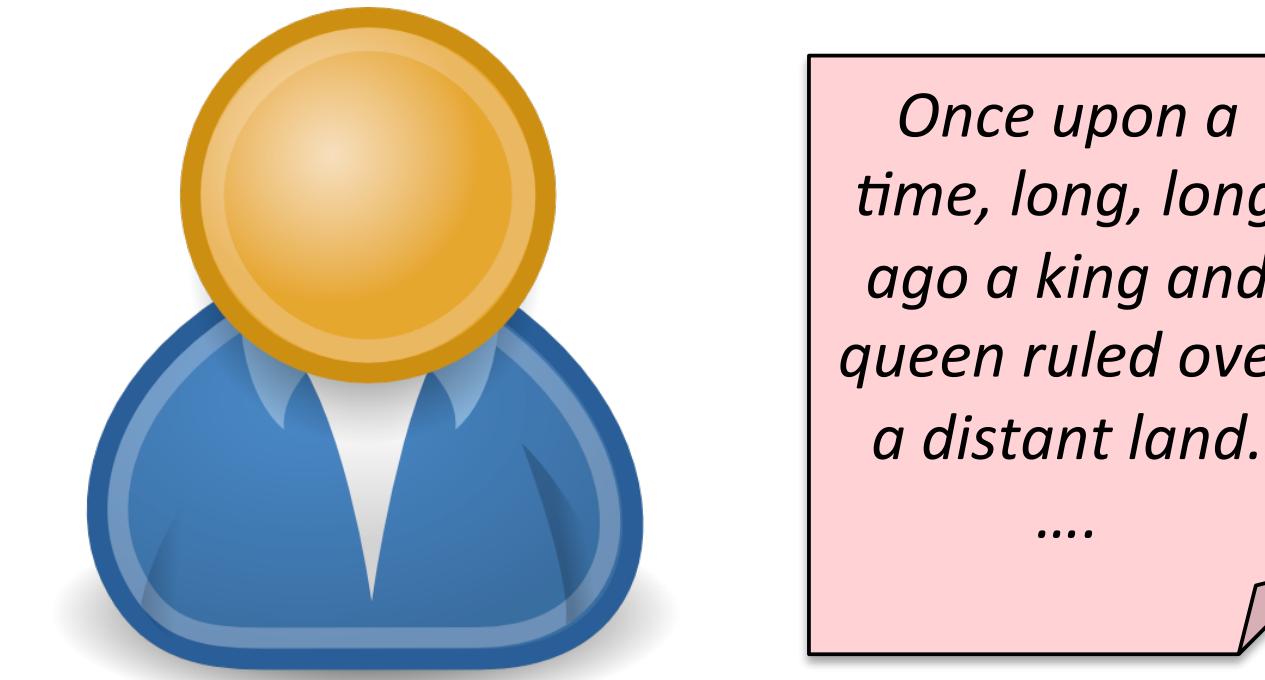
....

HDFS

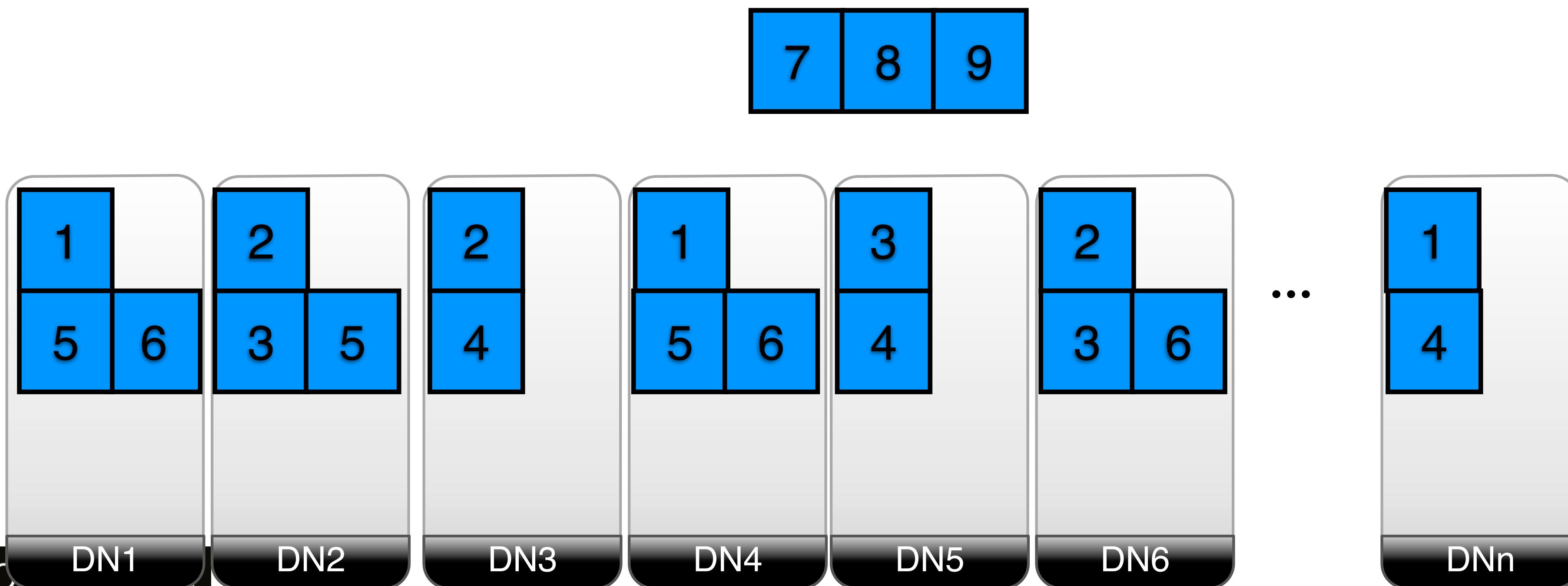
4	5	6
7	8	9



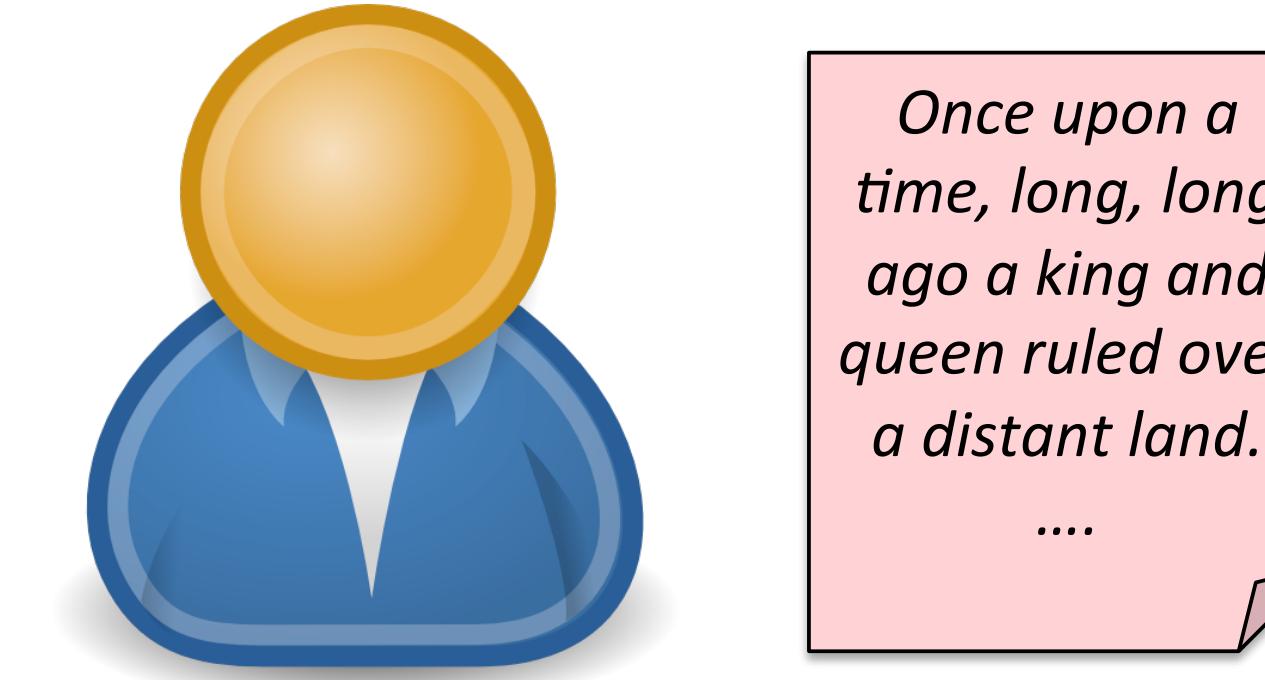
HDFS example



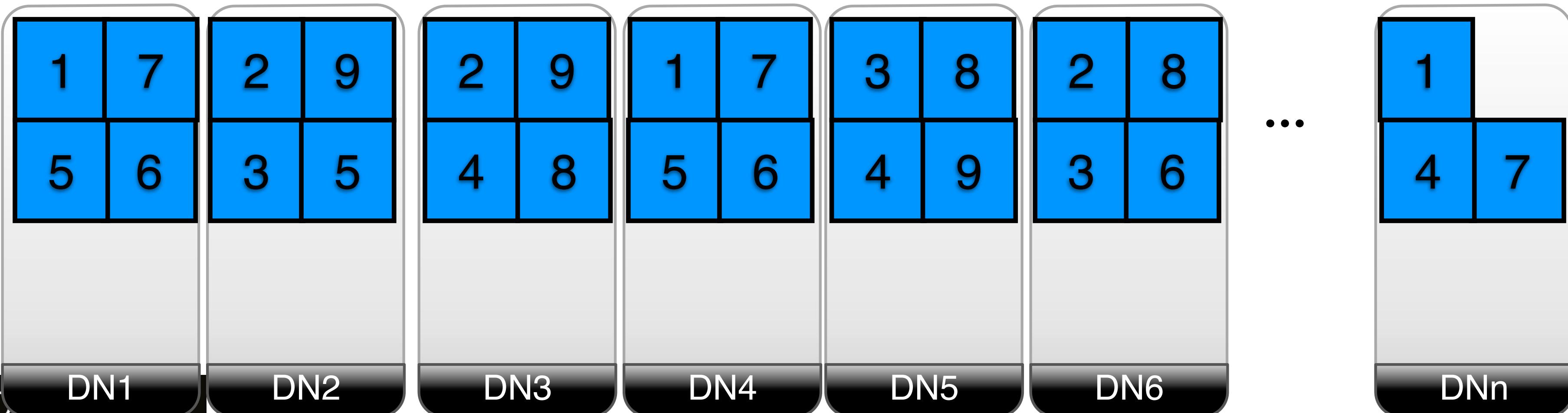
HDFS



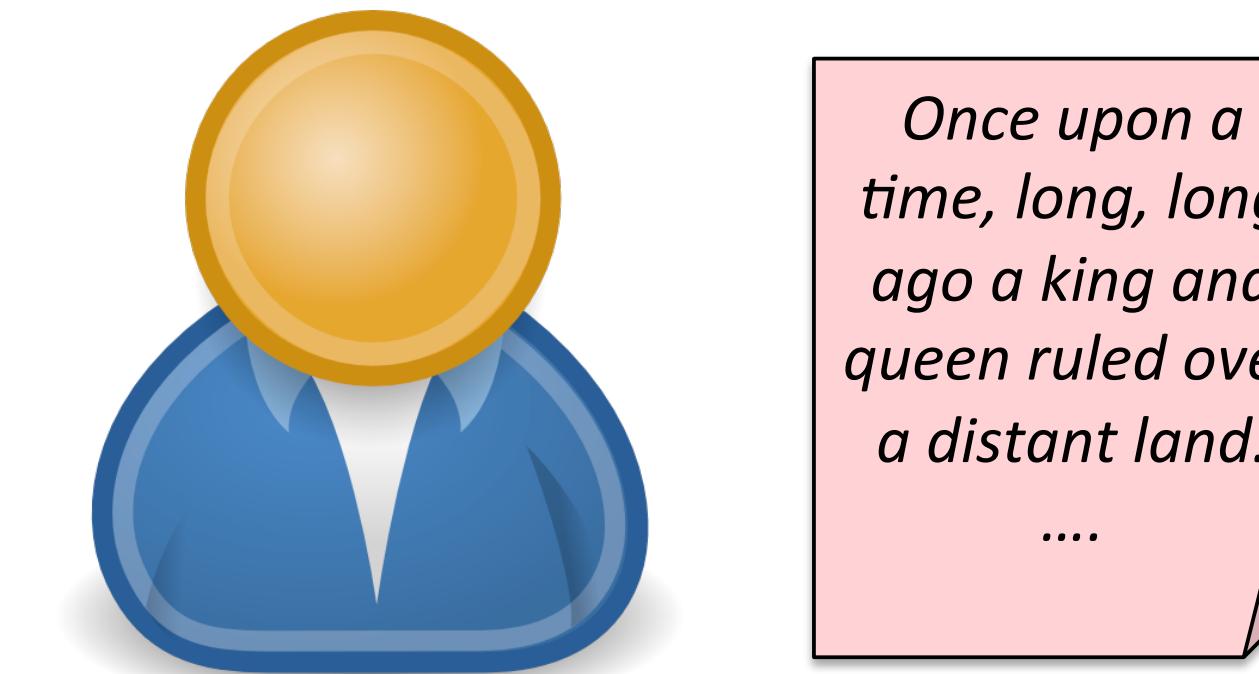
HDFS example



HDFS



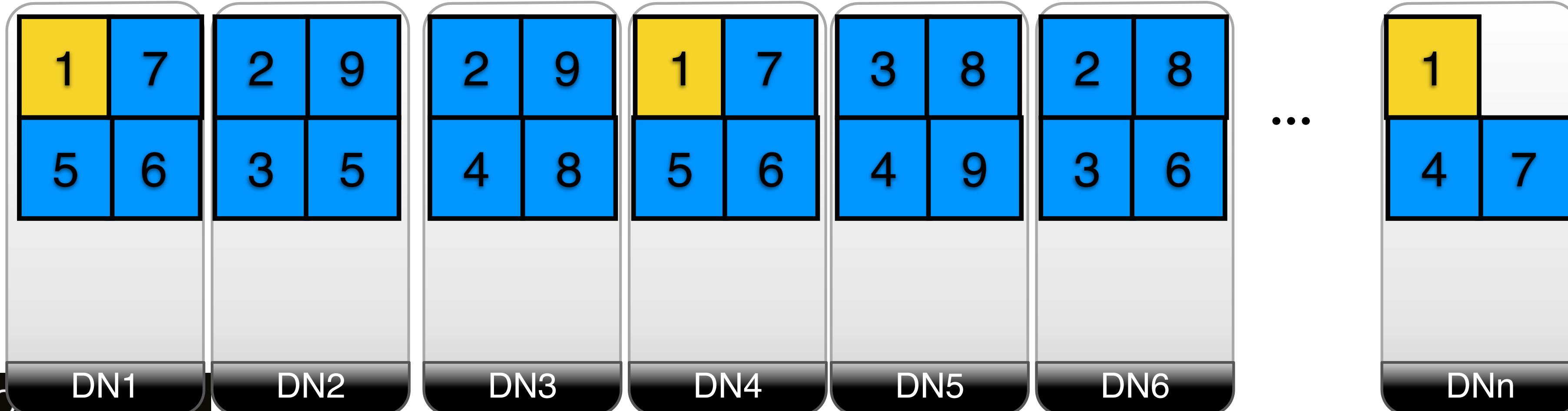
HDFS example



HDFS

partitioning

replication (by default factor 3)



HDFS example

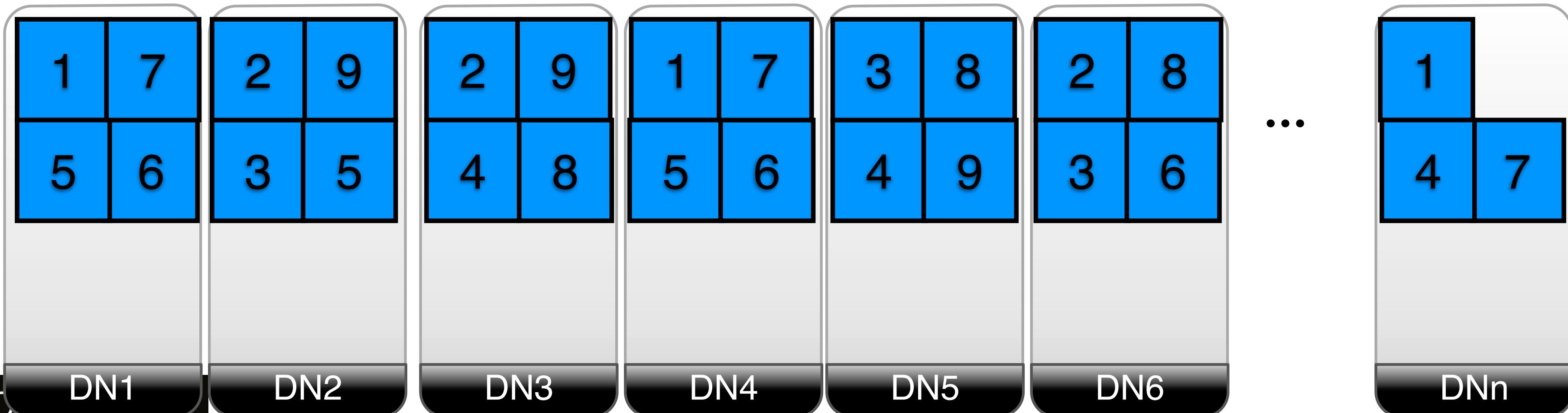


Fault-tolerance



*Once upon a
time, long, long
ago a king and
queen ruled over
a distant land.
....*

HDFS



HDFS example

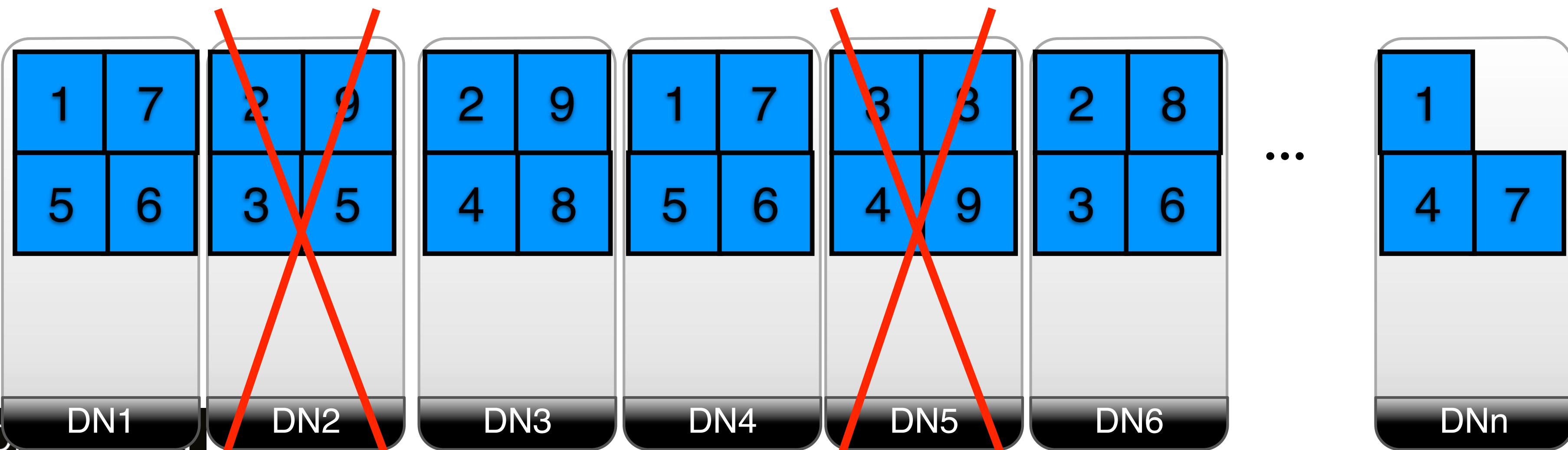


Fault-tolerance



Once upon a
time, long, long
ago a king and
queen ruled over
a distant land.
....

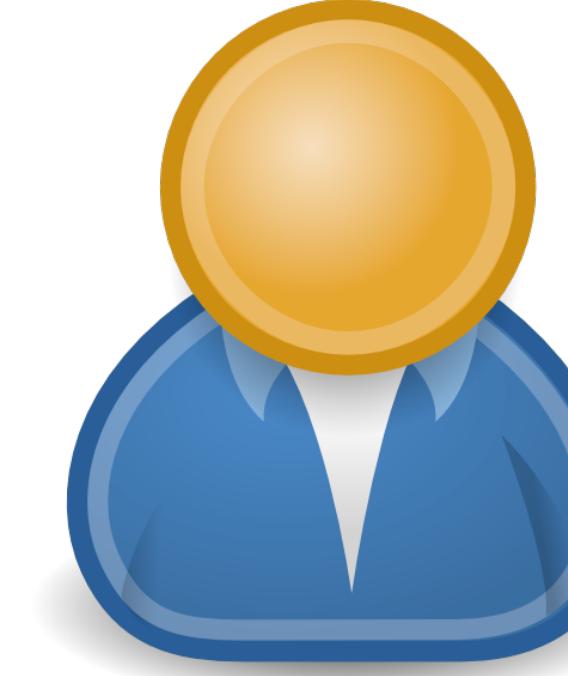
HDFS



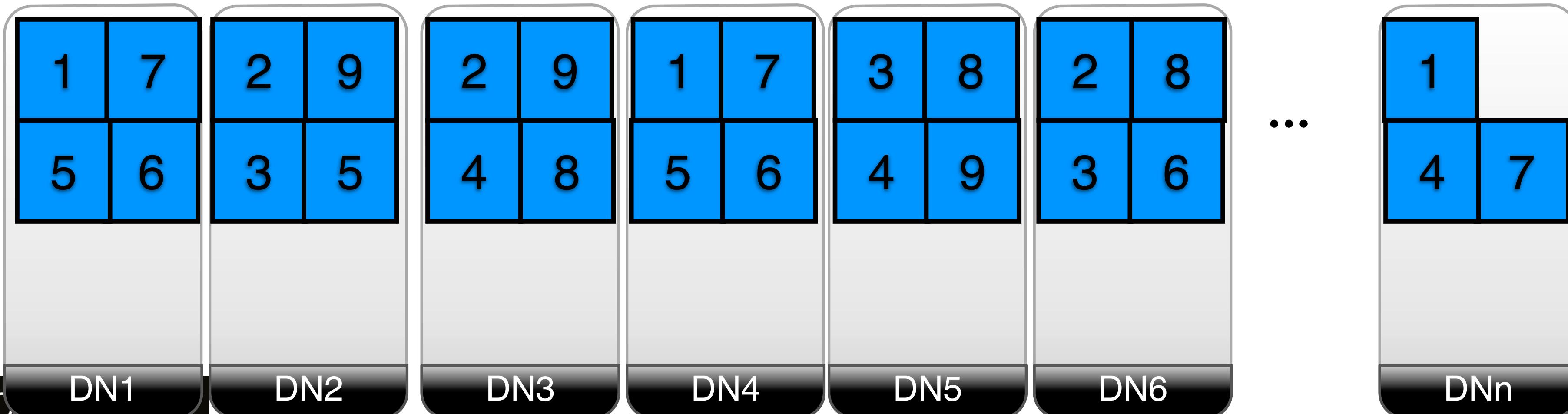
HDFS example



Load balancing



HDFS

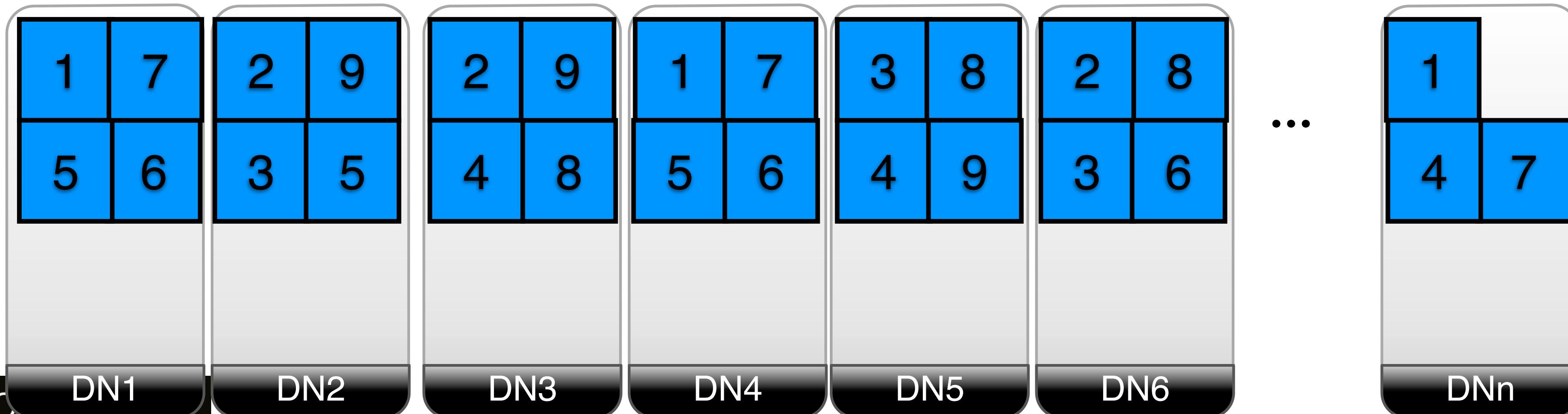


HDFS example

Load balancing

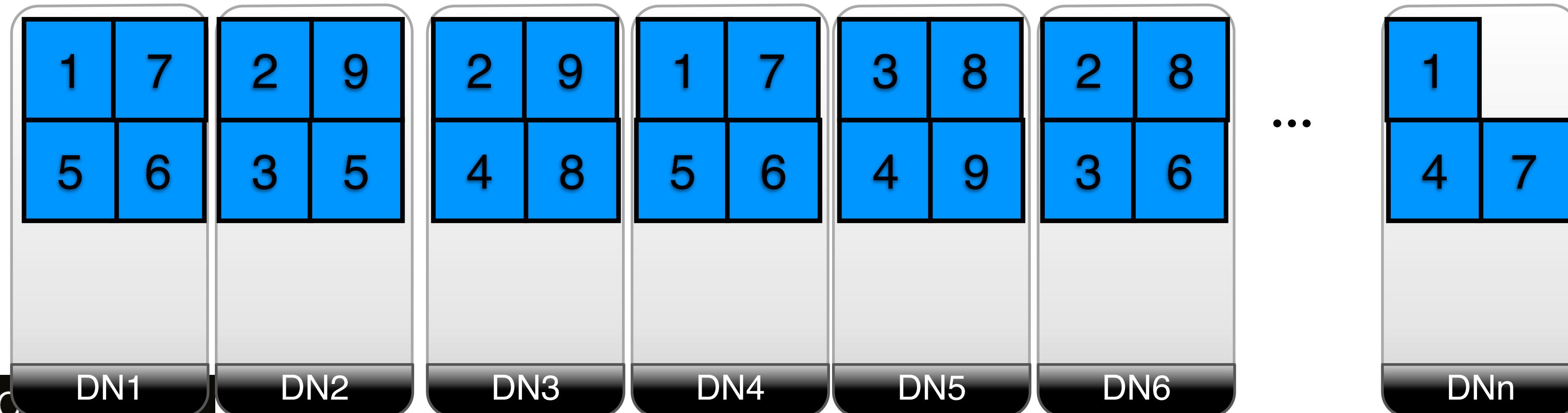
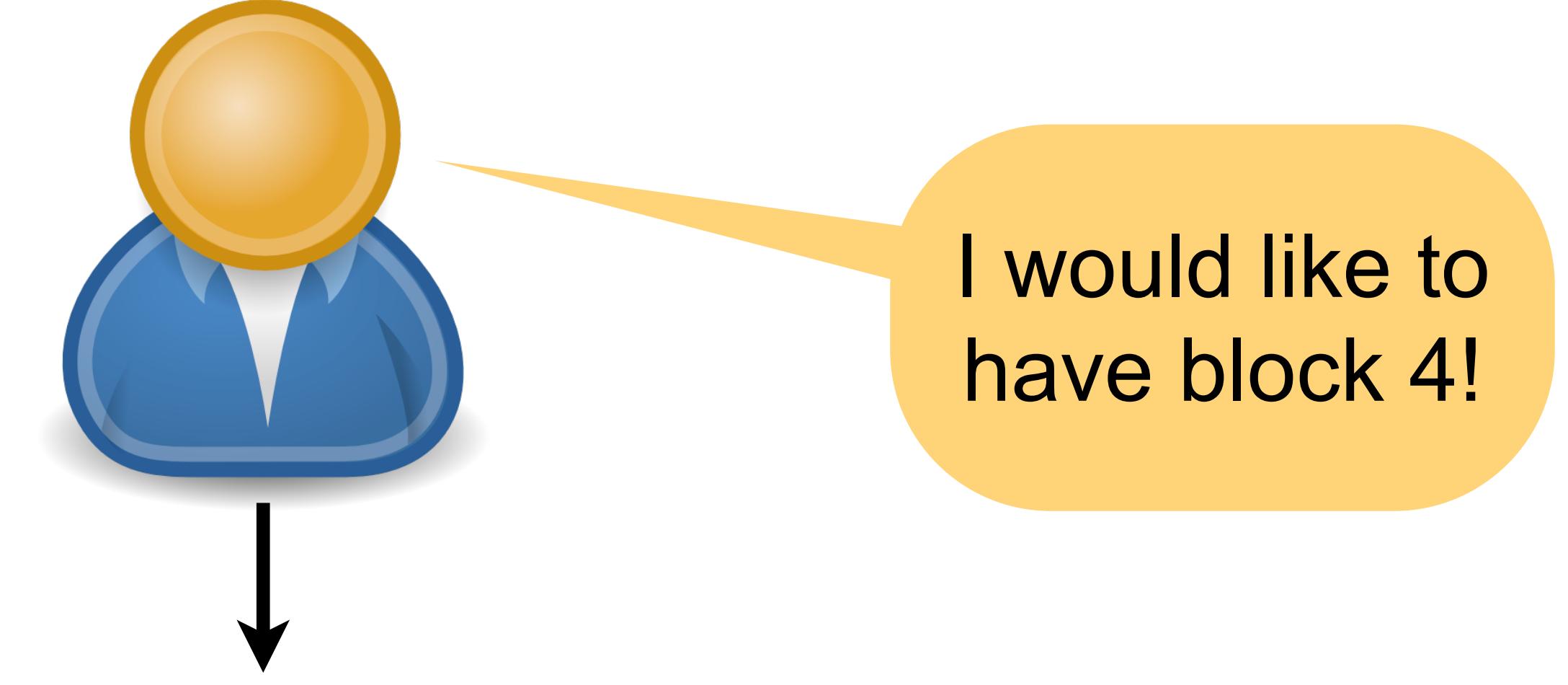


HDFS



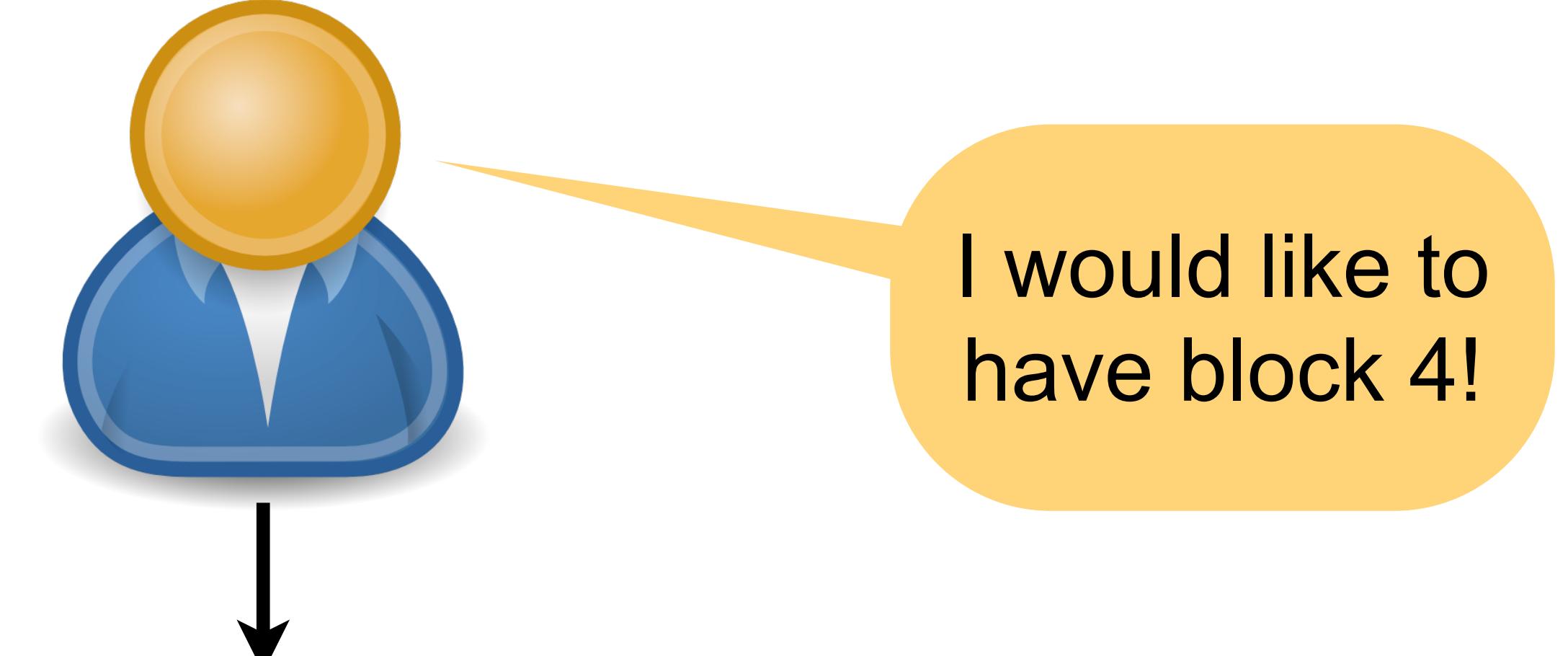
HDFS example

Load balancing

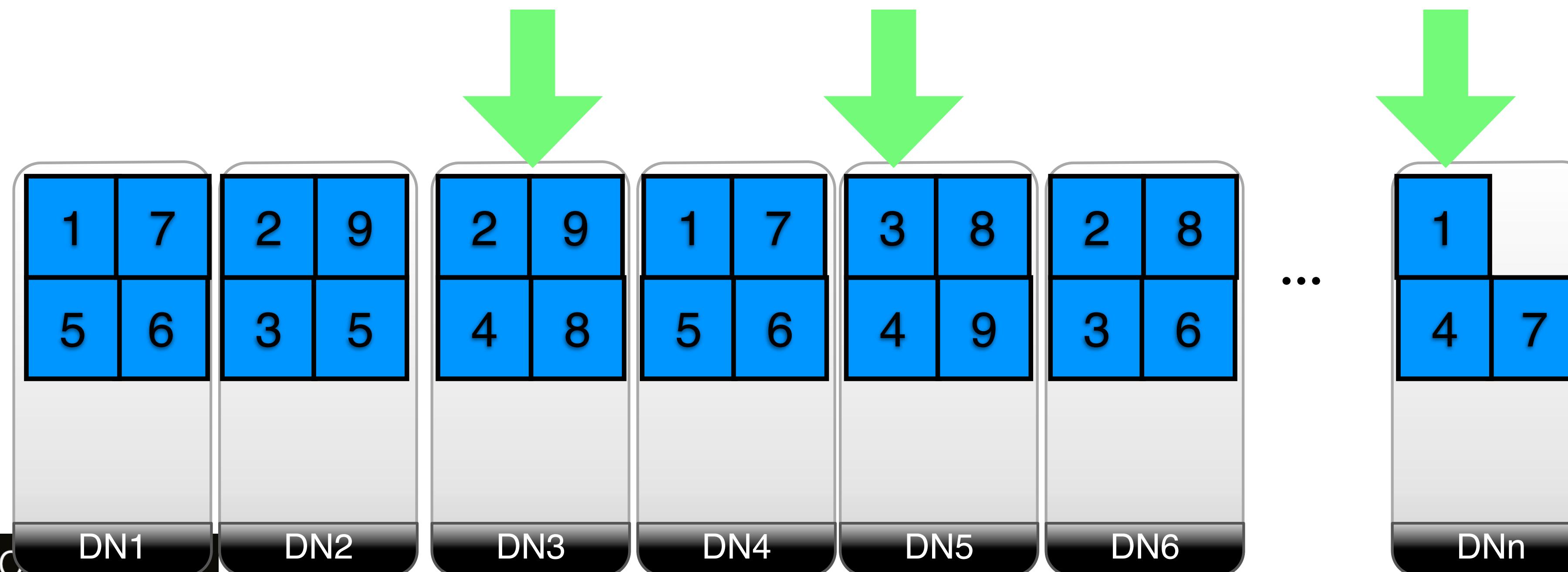


HDFS example

Load balancing



HDFS



HDFS example

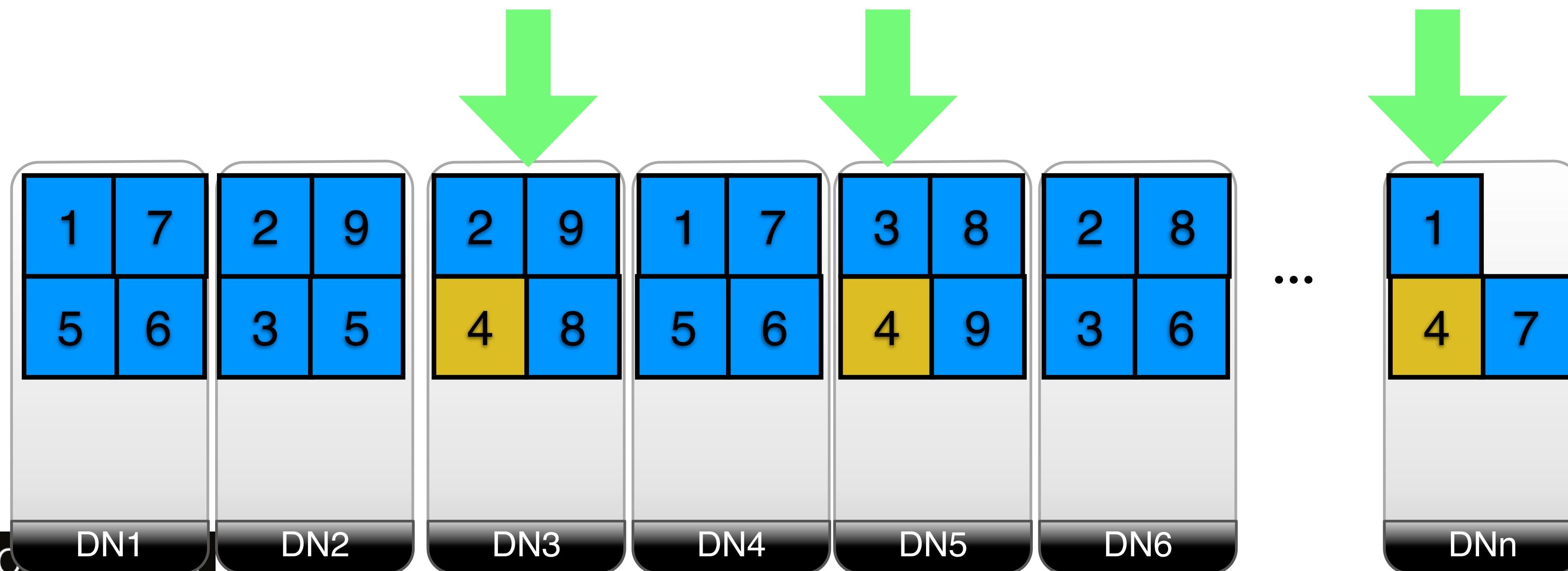


Load balancing



I would like to have block 4!

HDFS



Today's lecture

- ◆ Distributed file systems (HDFS)
- ◆ **Distributed data processing paradigms (MapReduce)**
- ◆ Distributed data processing platforms (Hadoop, Spark)



What is MapReduce and Hadoop

- ◆ **MapReduce:** A programming model
 - ◆ Functional style
 - ◆ Users specify Map and Reduce functions
 - ◆ Expressive
- ◆ **Hadoop:** An associated implementation
 - ◆ Automatically parallelized
 - ◆ Automatic partitioning, execution, failure handling, load balancing, communication
 - ◆ Can handle very large datasets of clusters of commodity computers

MapReduce in a nutshell (I)

- ◆ Framework
 - ◆ Read lots of data
 - ◆ **Map**: process a data item / record
 - ◆ **Sort and shuffle**
 - ◆ **Reduce**: aggregate, summarize, filter, transform
 - ◆ Write results
- ◆ Map and Reduce are **user-specified** → used to model given problem

MapReduce in a nutshell (II)

- ♦ Simple API

- ♦ **Map** (*key, value*) —> {*ikey, ivalue*}
 - ♦ **Reduce** (*ikey, {ivalue}s*) —> (*key', value'*)

- ♦ **Map phase**: independent processes (mappers) which run **in parallel**

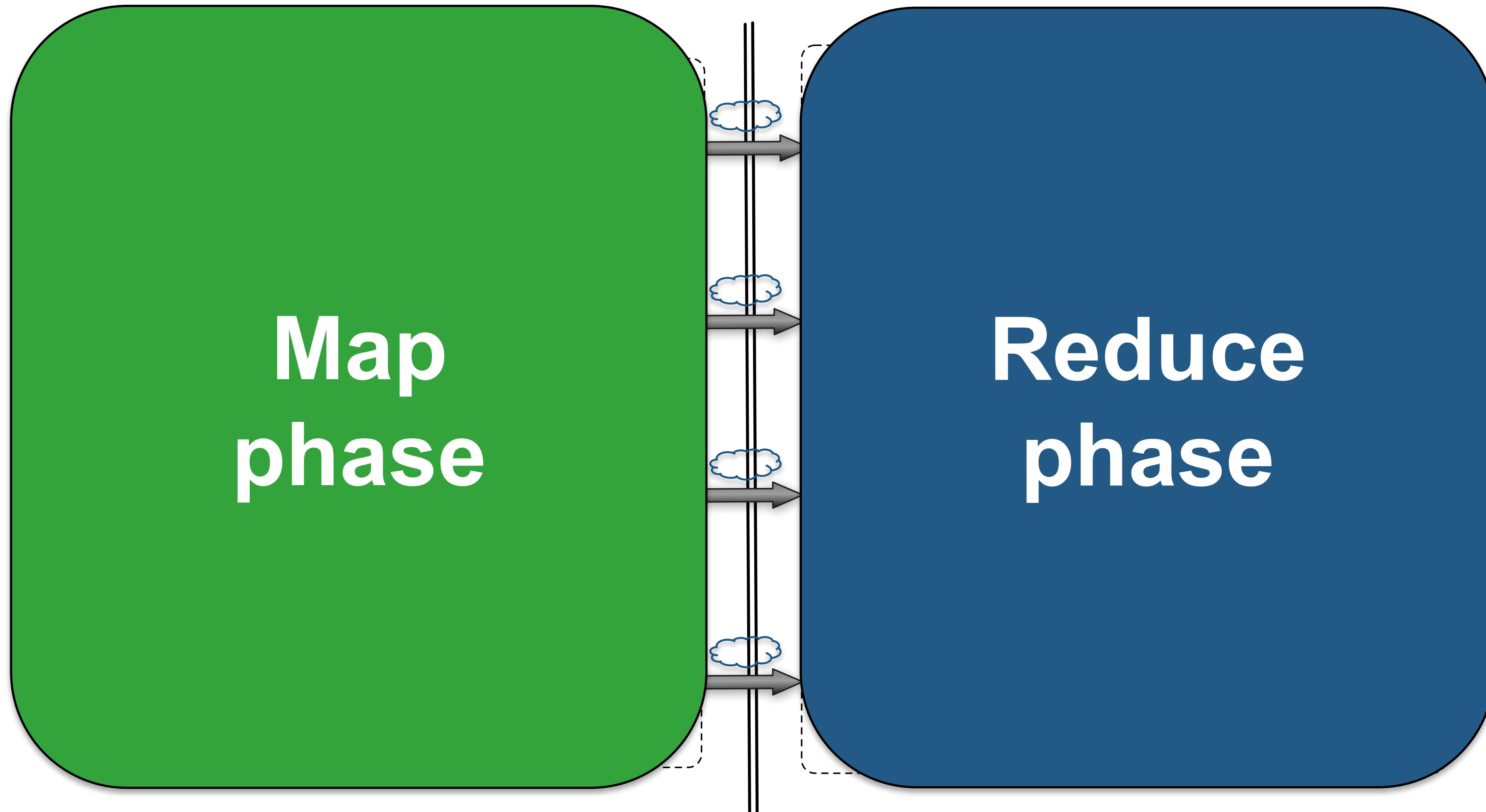
- ♦ operate on the chunks (blocks) of input data
 - ♦ output intermediate results

- ♦ **Shuffle phase**: Intermediate results are shuffled through the network

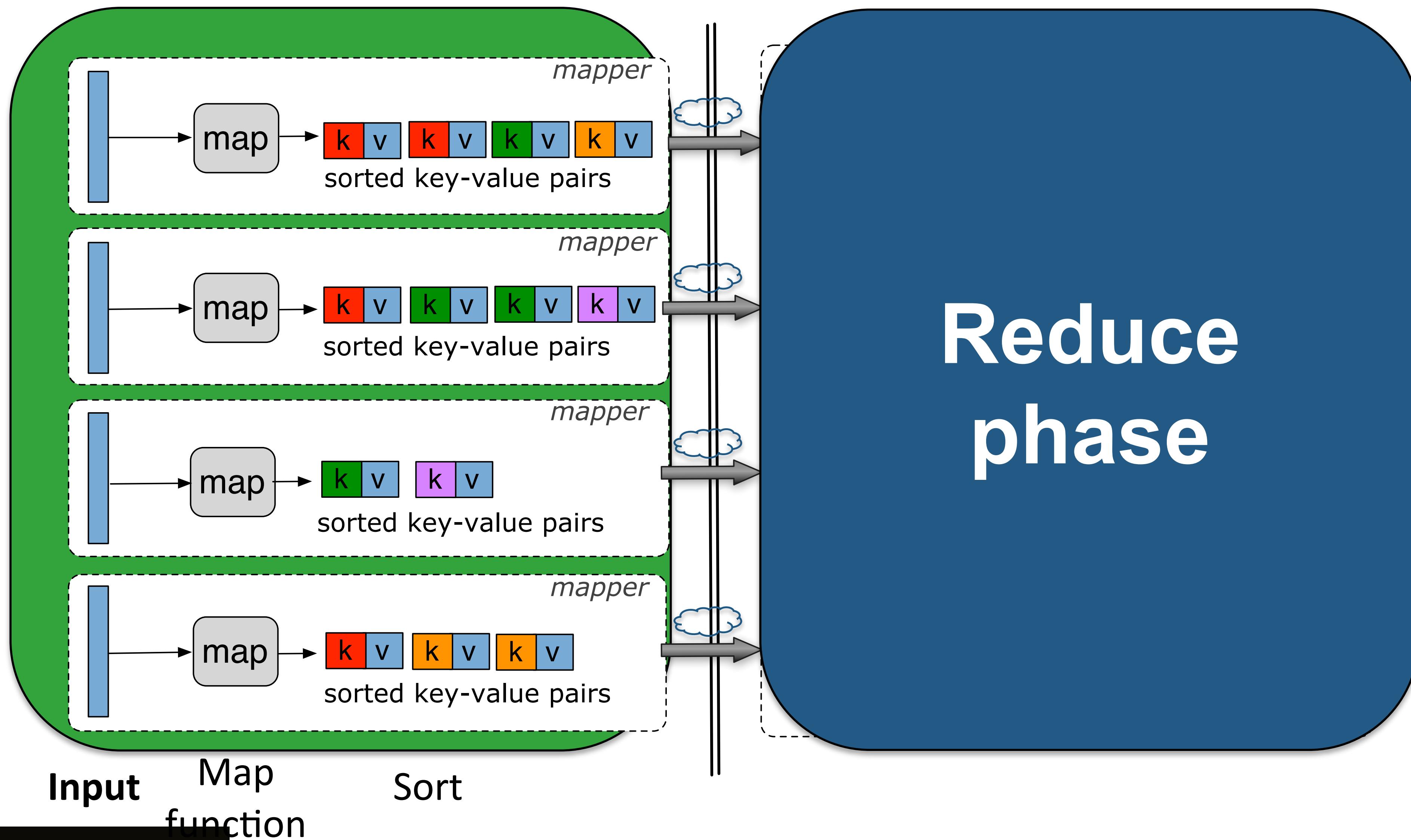
- ♦ **Reduce phase**: independent processes (reducers) which run **in parallel**

- ♦ group intermediate results of the map phase
 - ♦ operate on the groups
 - ♦ output final results

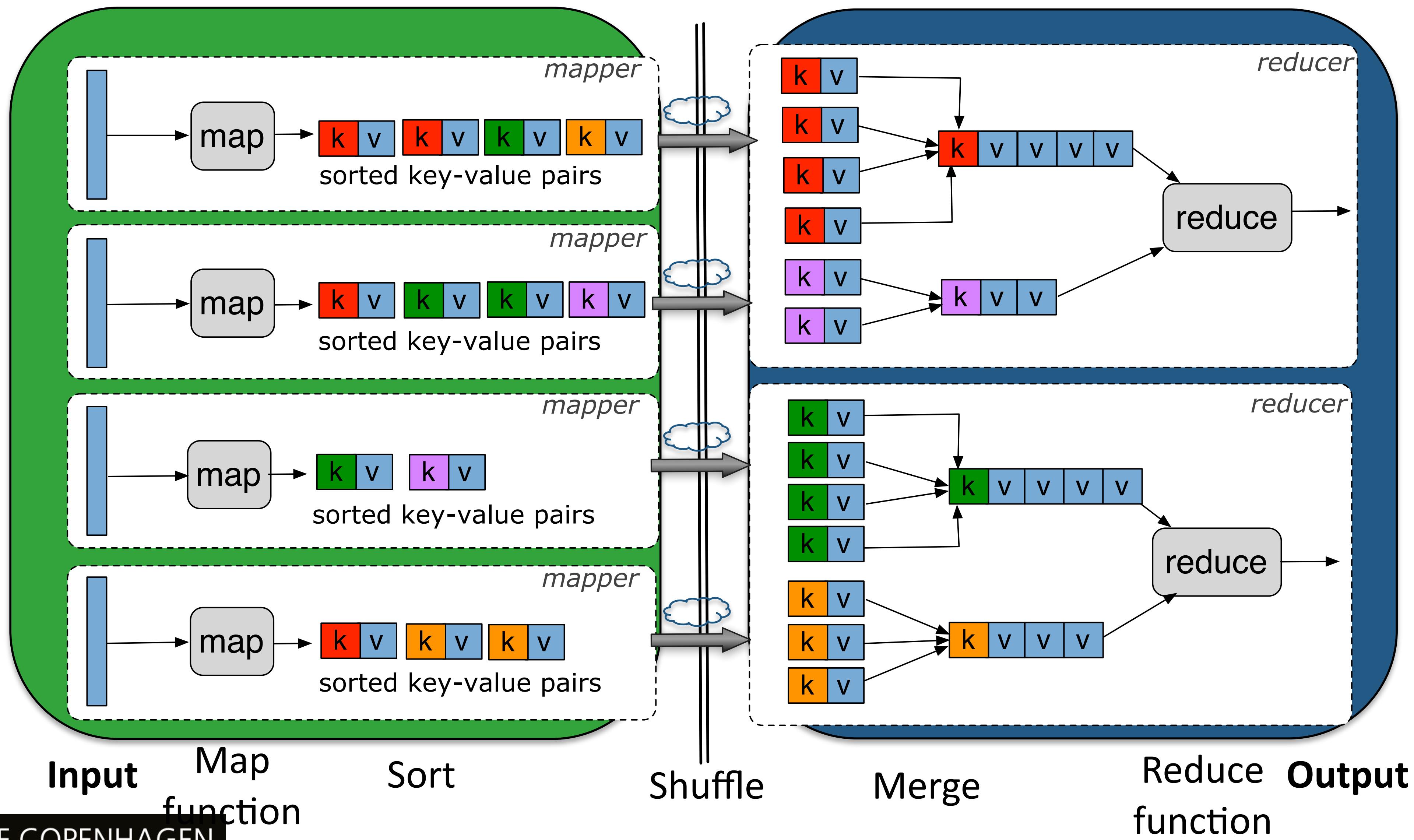
MapReduce illustration



MapReduce illustration



MapReduce illustration



Wordcount example

Map (*key, value*) —> {*ikey, ivalue*}

Reduce (*ikey, {ivalue}*) —> (*key', value'*)

Wordcount example

Map (*key, value*) \rightarrow {*ikey, ivalue*}

Reduce (*ikey, {ivalue}*) \rightarrow (*key', value'*)



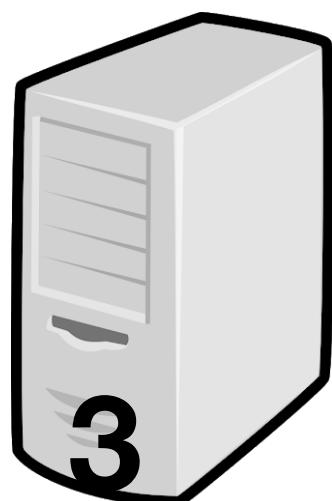
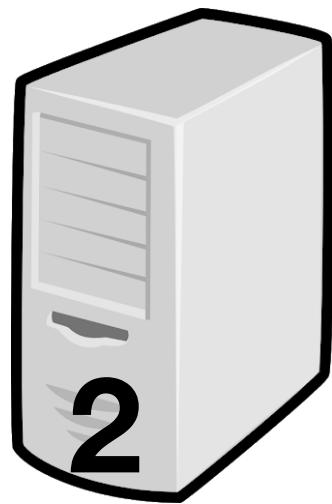
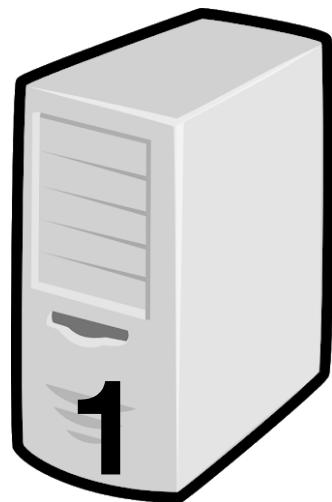
Map (*docID, text*) \rightarrow {*word, 1*}

Reduce (*word, {1, 1, ...}*) \rightarrow (*word, count*)

Wordcount example

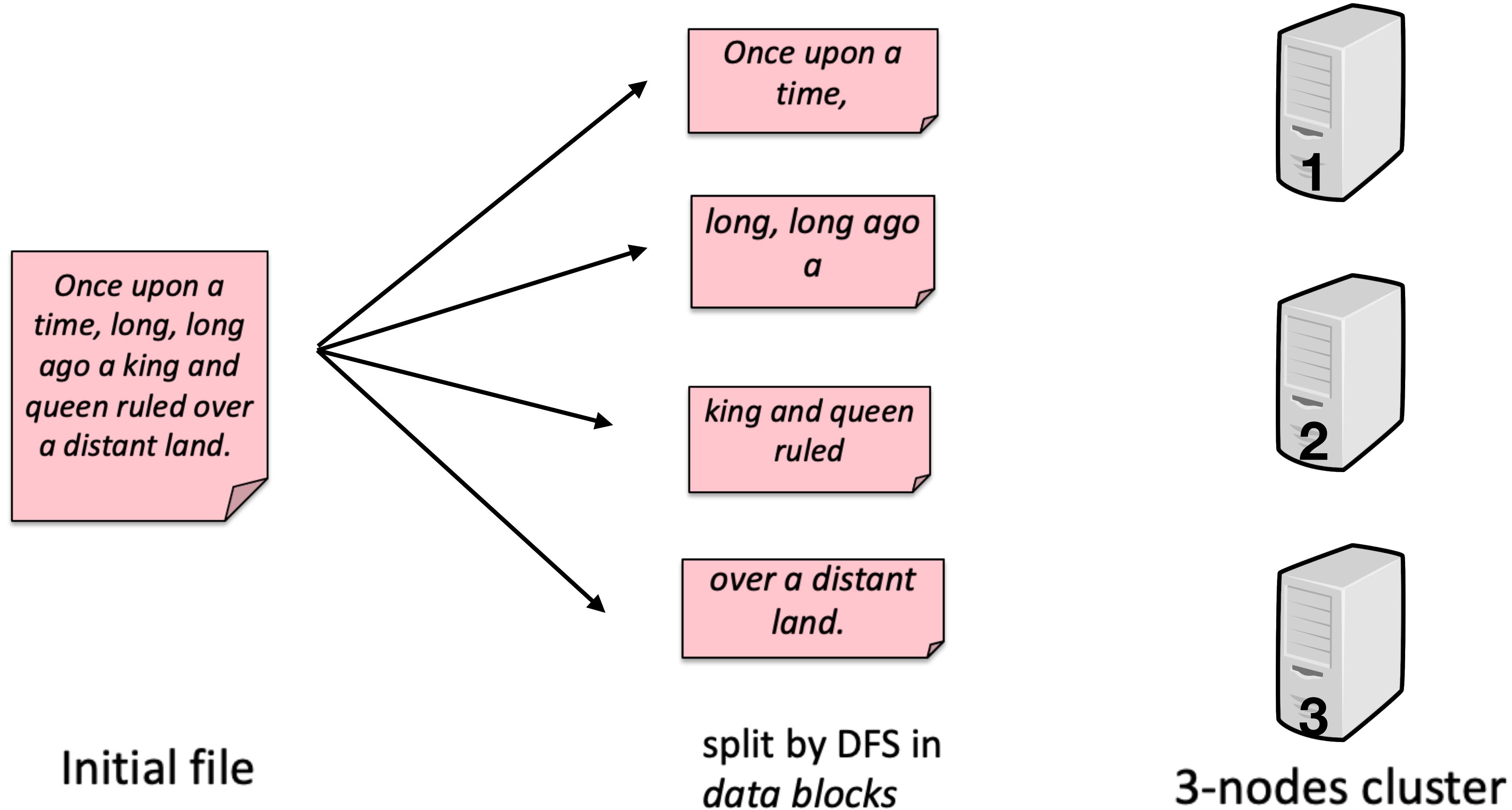
*Once upon a
time, long, long
ago a king and
queen ruled over
a distant land.*

Initial file



3-nodes cluster

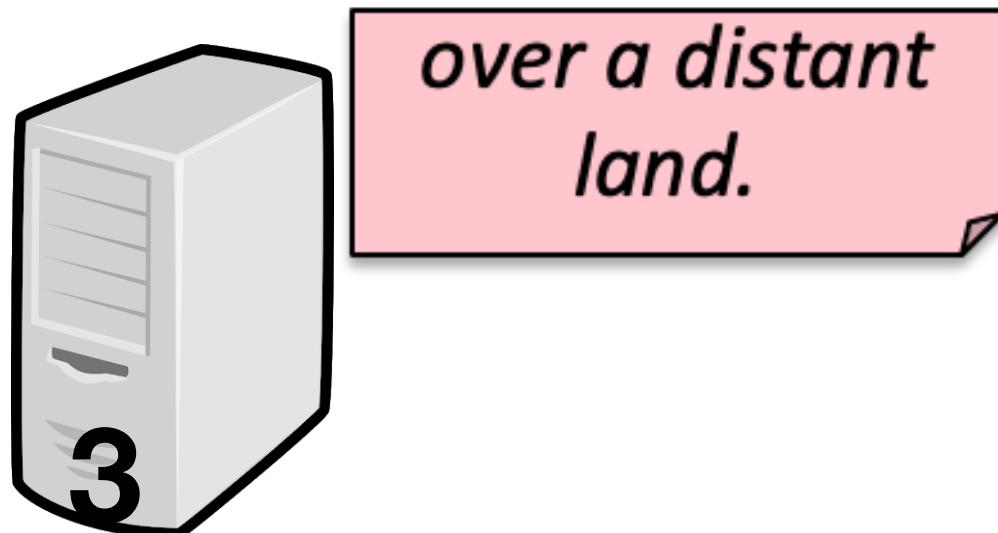
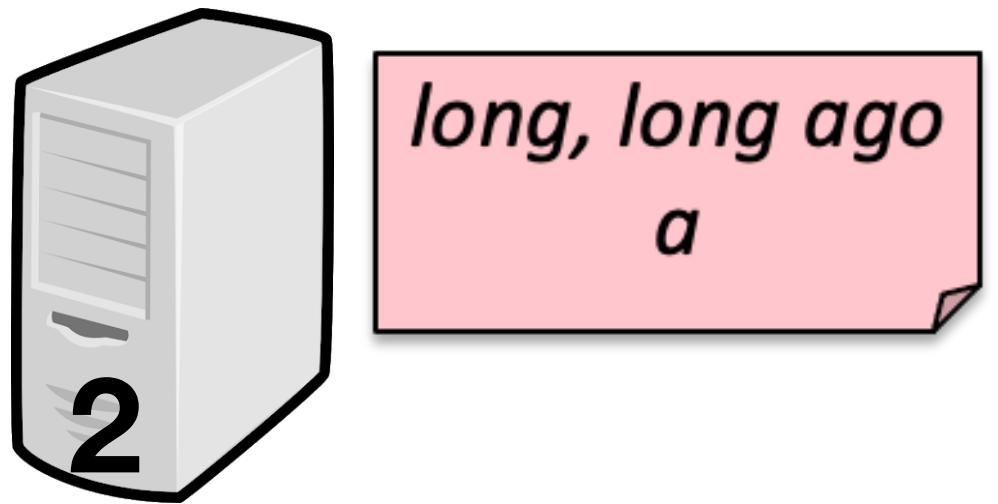
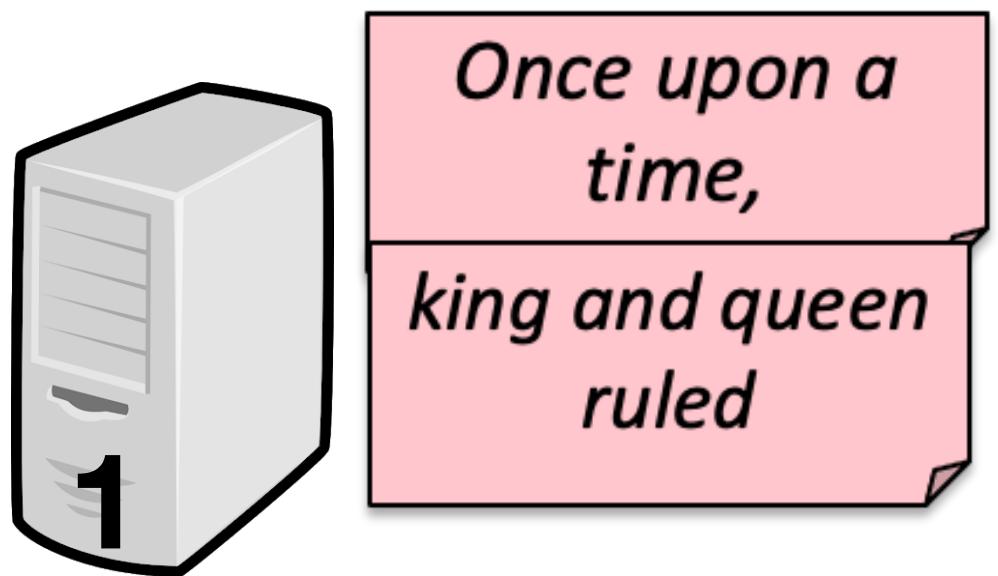
Wordcount example



Wordcount example

Once upon a time, long, long ago a king and queen ruled over a distant land.

Initial file



3-nodes cluster

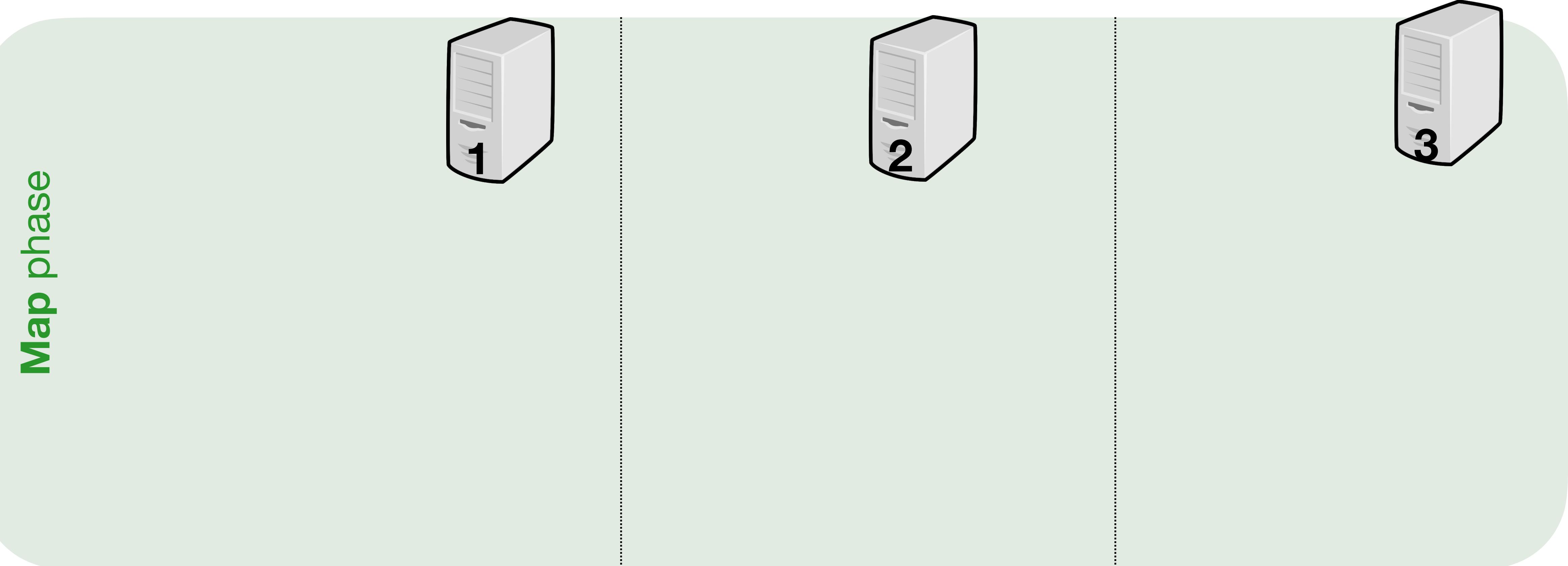
Wordcount example (Map phase)

Once upon a time,

king and queen ruled

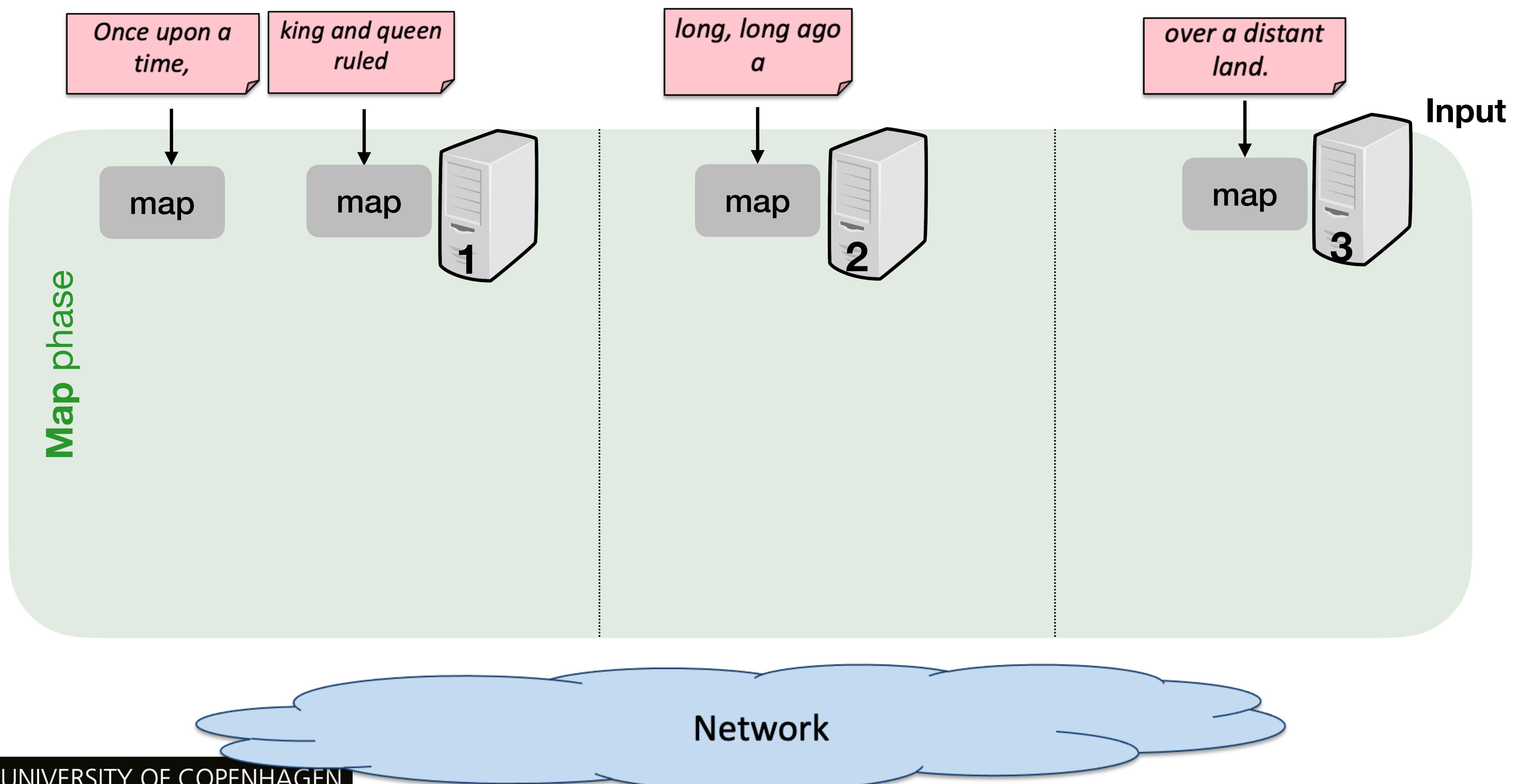
*long, long ago
a*

over a distant land.

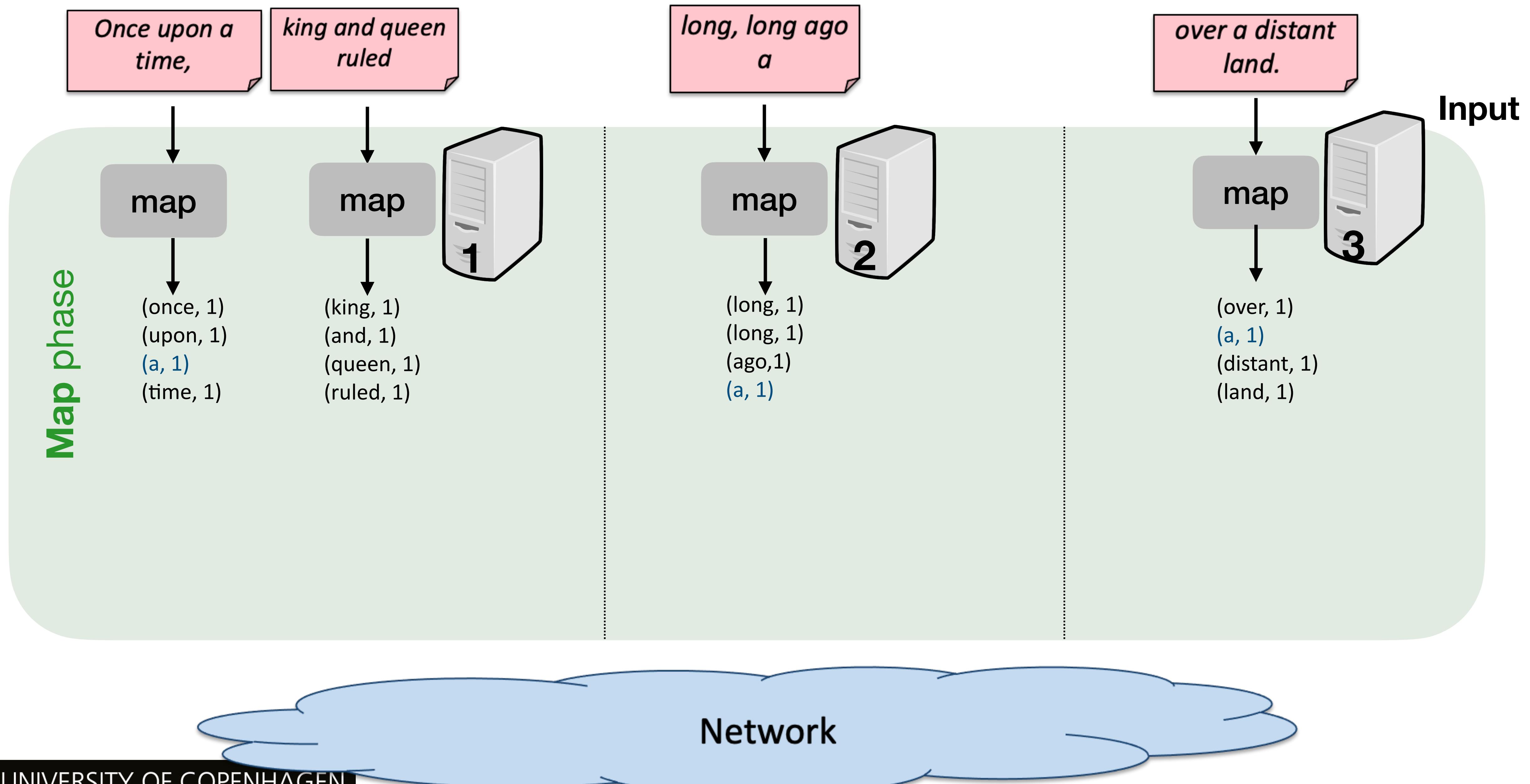


Network

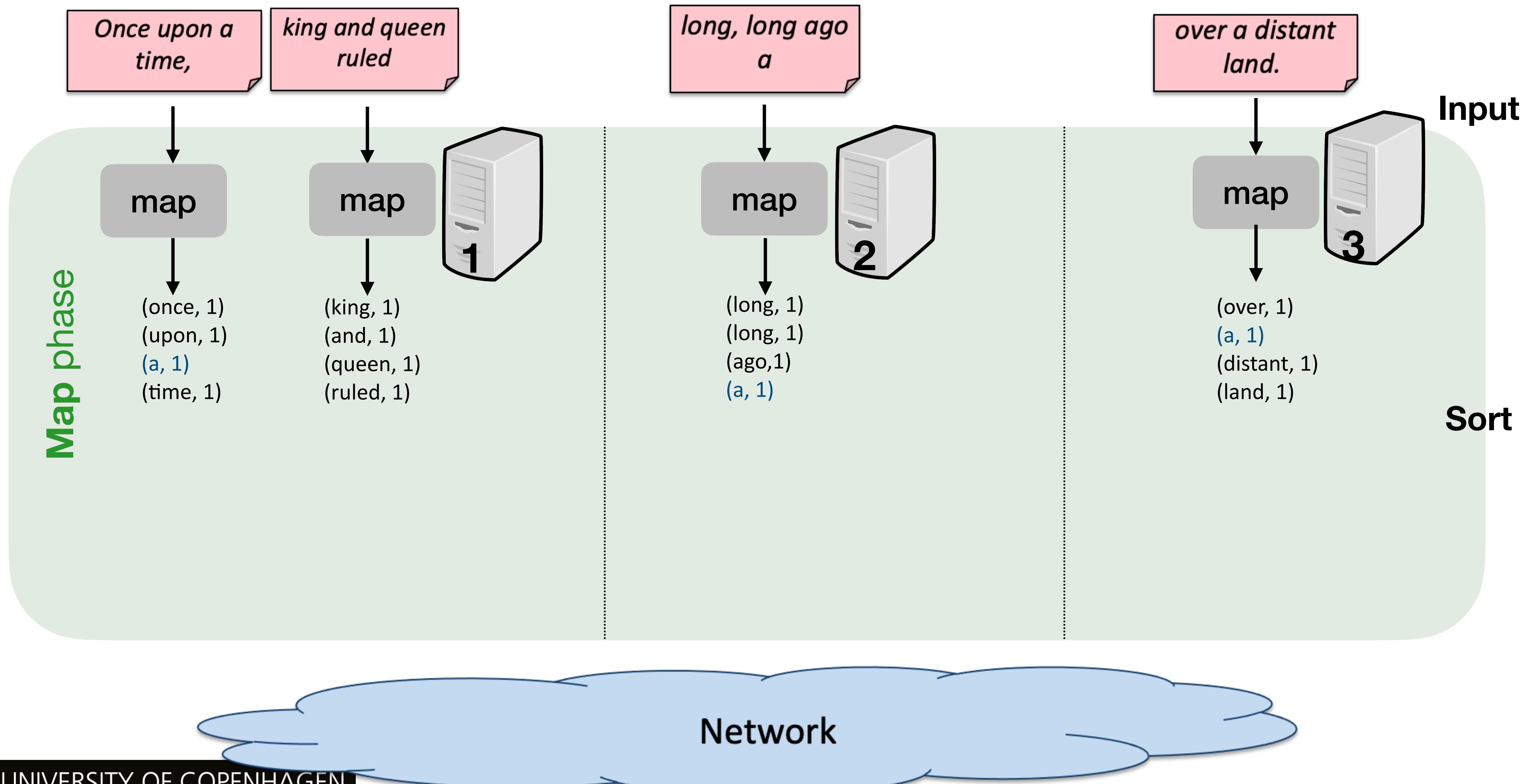
Wordcount example (Map phase)



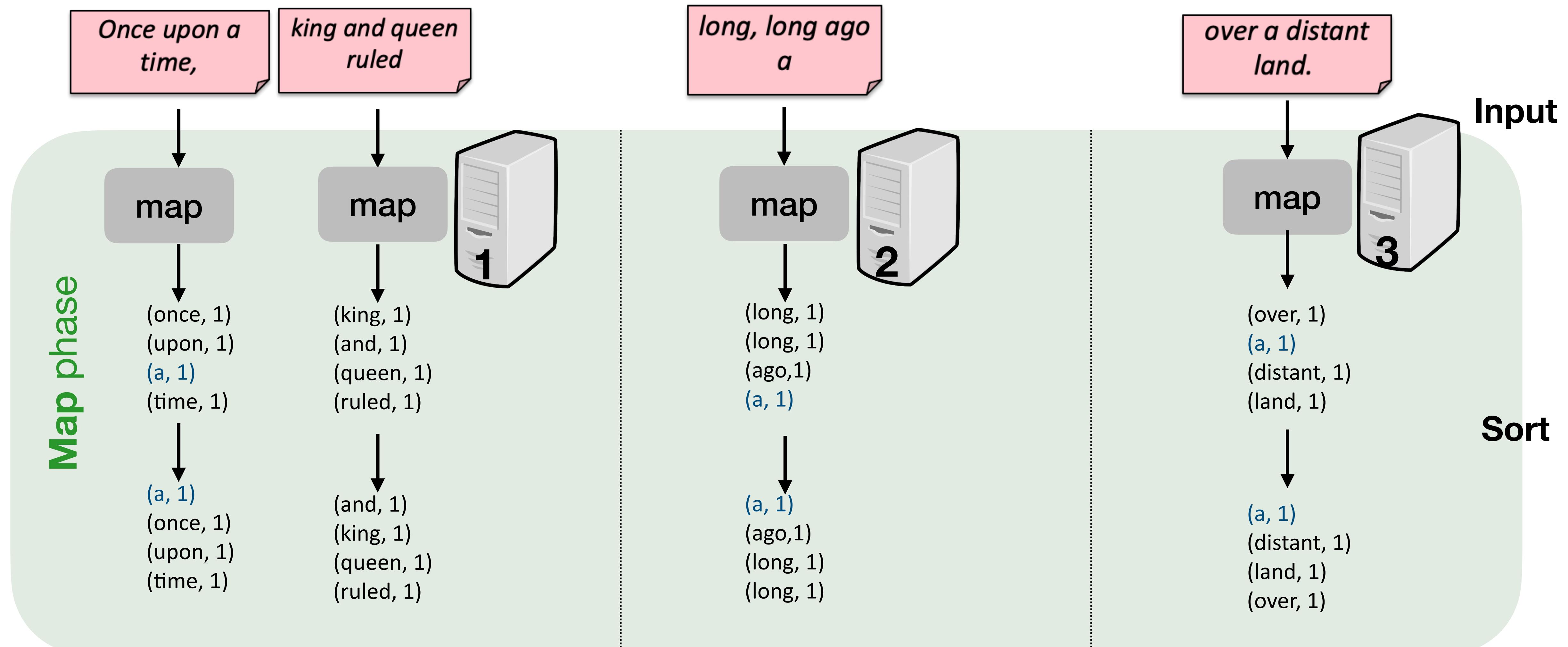
Wordcount example (Map phase)



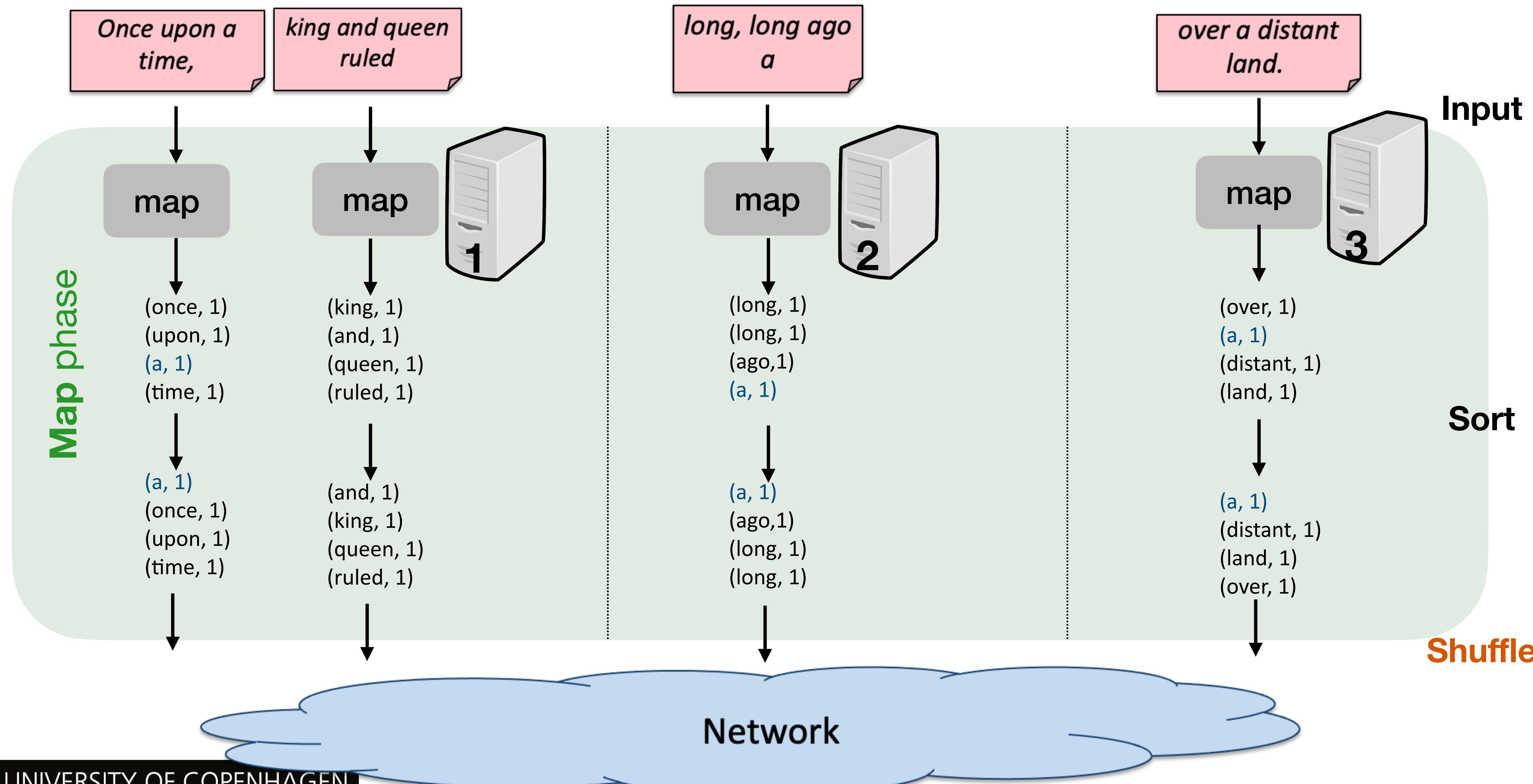
Wordcount example (Map phase)



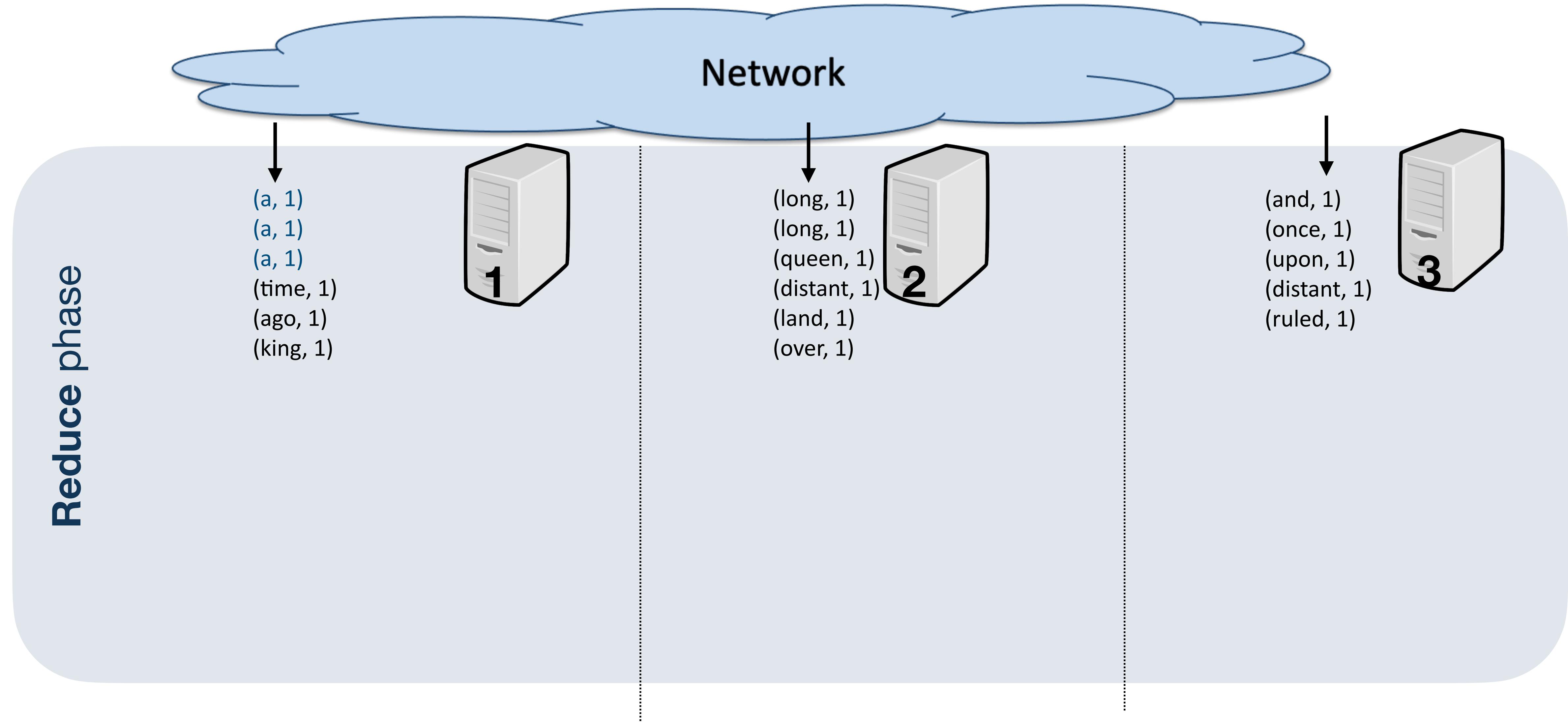
Wordcount example (Map phase)



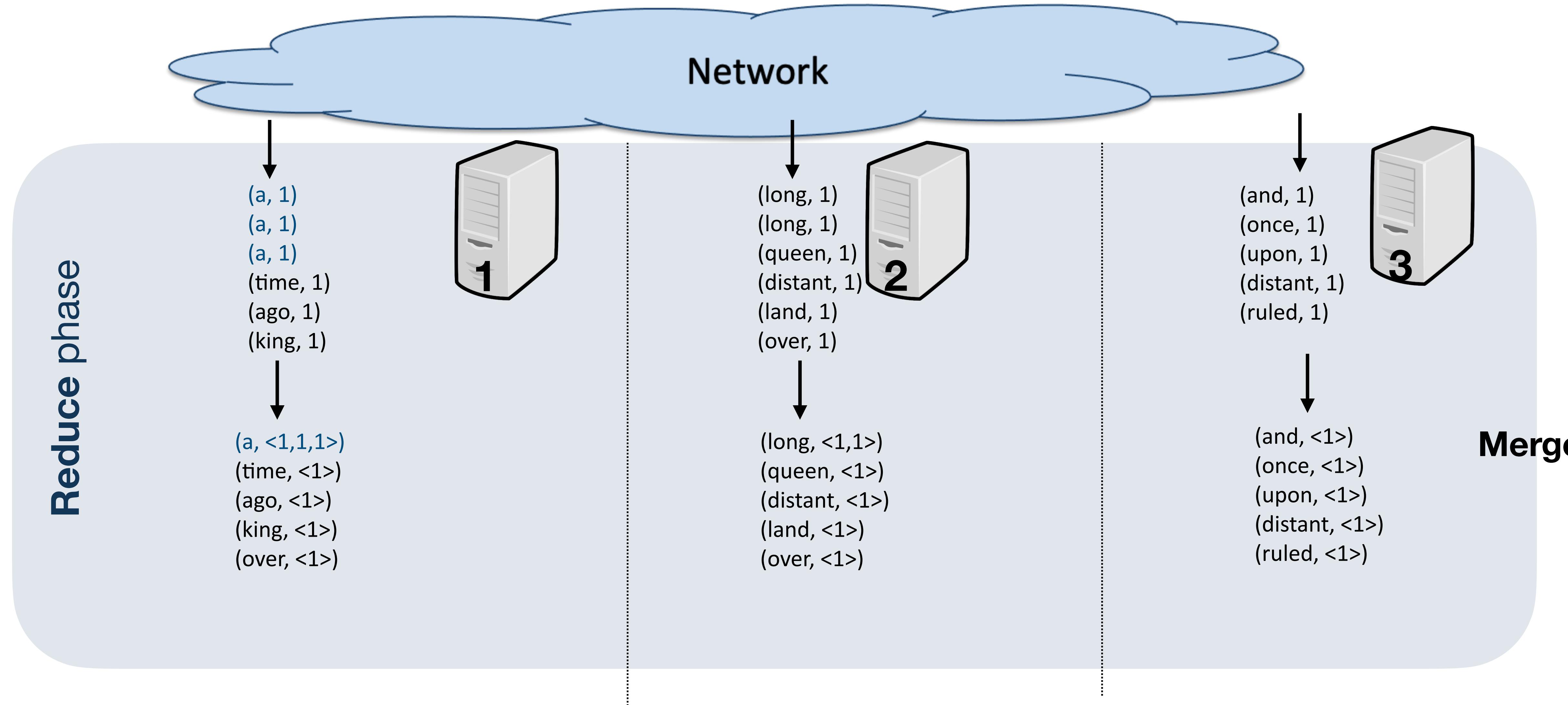
Wordcount example (Map phase)



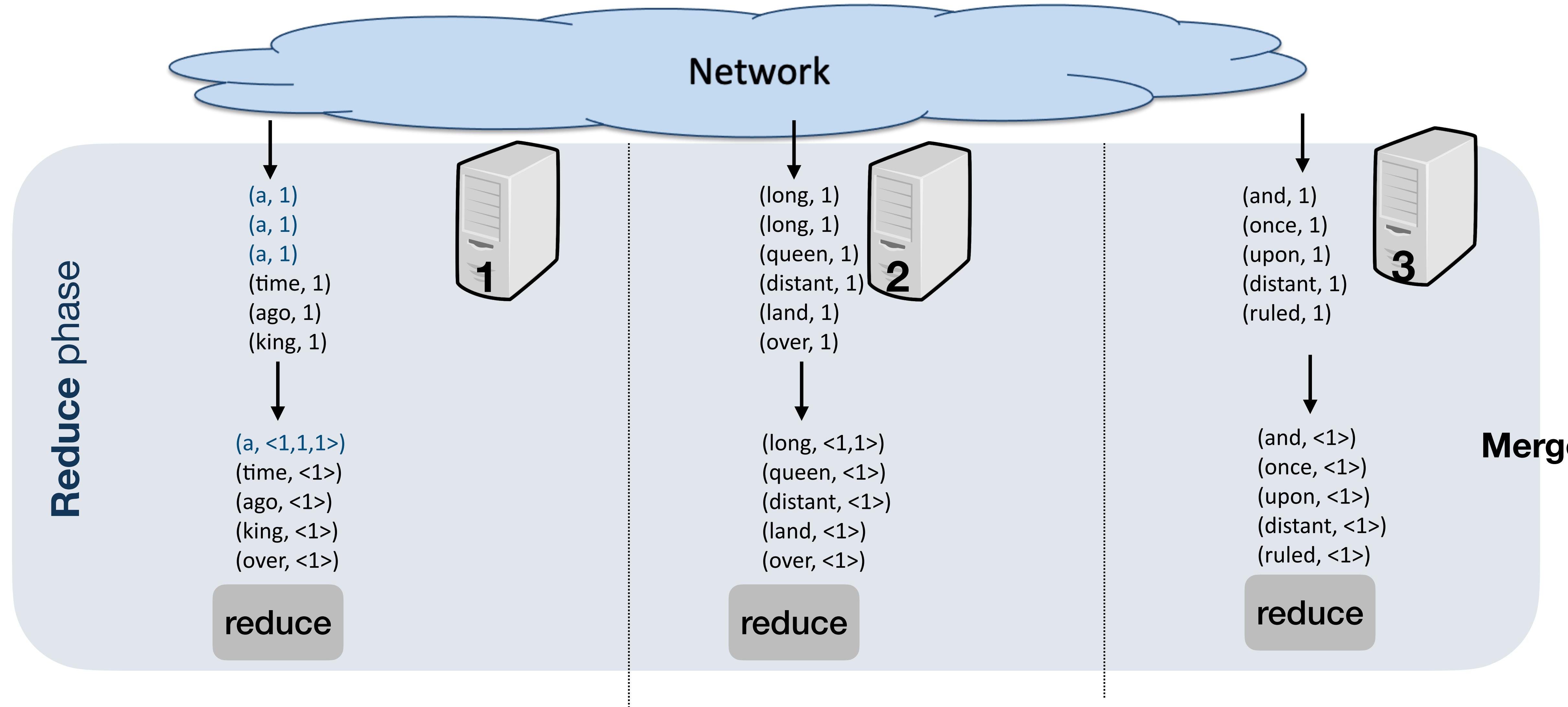
Wordcount example (Reduce phase)



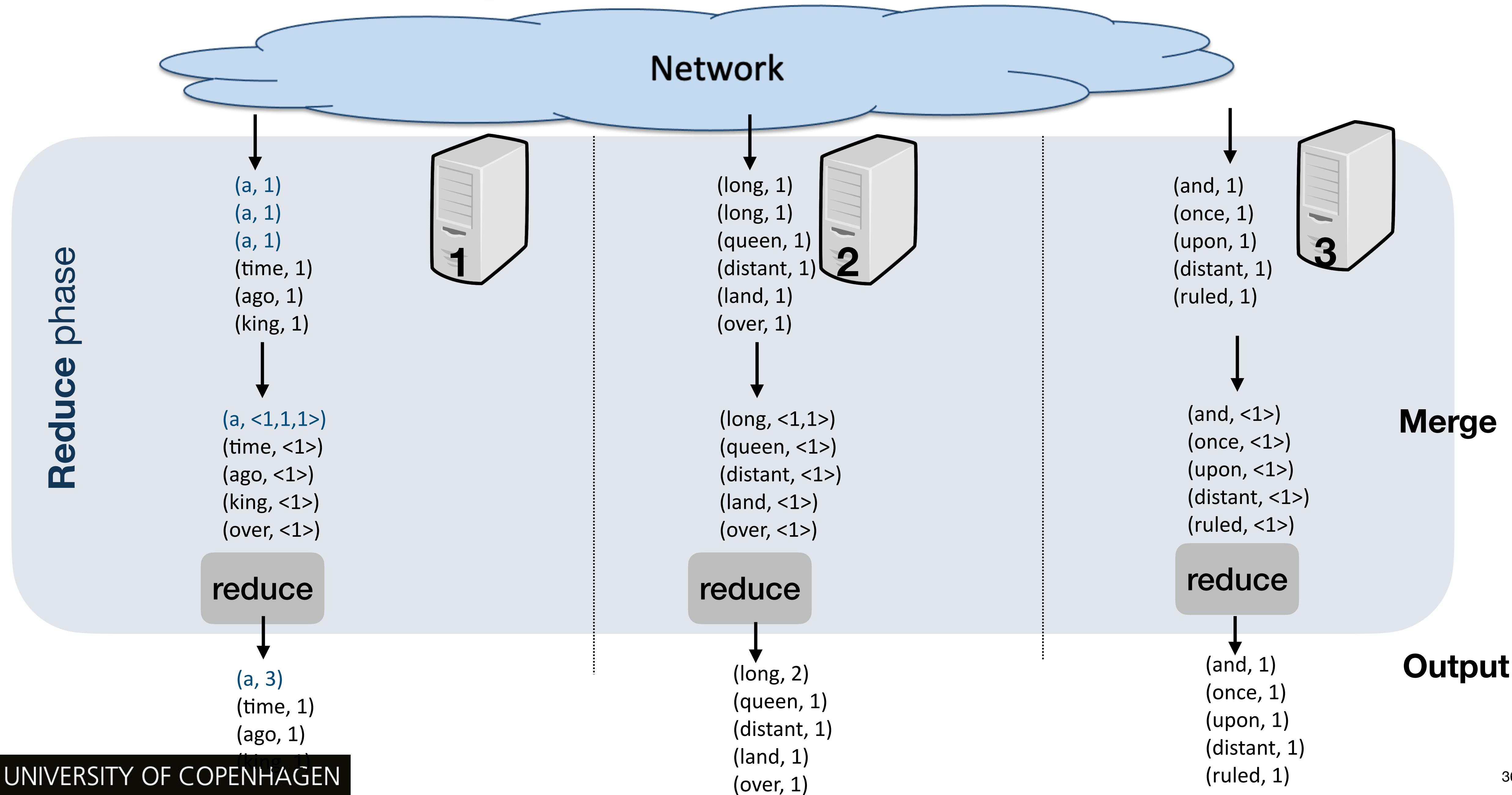
Wordcount example (Reduce phase)



Wordcount example (Reduce phase)



Wordcount example (Reduce phase)



WordCount example: pseudocode

WordCount example: pseudocode

```
Map(Integer key, String text)
  foreach word w in text:
    emit (w, 1)
```

WordCount example: pseudocode

Map(Integer key, String text)

foreach word w **in** text:

emit (w, 1)

Reduce(String word, Iterator<Integer> counts)

int result = 0

foreach count **in** counts:

 result += count

emit (word, result)

Hadoop/MapReduce Wordcount code :)

```
1 package org.myorg;
2
3 import java.io.IOException;
4 import java.util.*;
5
6 import org.apache.hadoop.fs.Path;
7 import org.apache.hadoop.conf.*;
8 import org.apache.hadoop.io.*;
9 import org.apache.hadoop.mapreduce.*;
10 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
11 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
12 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
13 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
14
15 public class WordCount {
16
17     public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
18         private final static IntWritable one = new IntWritable(1);
19         private Text word = new Text();
20
21         public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
22             String line = value.toString();
23             StringTokenizer tokenizer = new StringTokenizer(line);
24             while (tokenizer.hasMoreTokens()) {
25                 word.set(tokenizer.nextToken());
26                 context.write(word, one);
27             }
28         }
29     }
30
31     public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
32
33         public void reduce(Text key, Iterable<IntWritable> values, Context context)
34             throws IOException, InterruptedException {
35             int sum = 0;
36             for (IntWritable val : values) {
37                 sum += val.get();
38             }
39             context.write(key, new IntWritable(sum));
40         }
41     }
42
43     public static void main(String[] args) throws Exception {
44         Configuration conf = new Configuration();
45
46         Job job = new Job(conf, "wordcount");
47
48         job.setOutputKeyClass(Text.class);
49         job.setOutputValueClass(IntWritable.class);
50
51         job.setMapperClass(Map.class);
52         job.setReducerClass(Reduce.class);
53
54         job.setInputFormatClass(TextInputFormat.class);
55         job.setOutputFormatClass(TextOutputFormat.class);
56
57         FileInputFormat.addInputPath(job, new Path(args[0]));
58         FileOutputFormat.setOutputPath(job, new Path(args[1]));
59
60         job.waitForCompletion(true);
61     }
62
63 }
```



Only 63 lines for a distributed/
parallel application!

Parallelization in MapReduce

- ◆ Application of user-defined function f in `map()` on a data item often not influenced by computations on other data items
- ◆ **Embarassingly parallel**
- ◆ Implies that we can re-order or parallelize the execution
- ◆ Note: not true when f has side-effects or state



This property forms the basis for MapReduce!

Hadoop/MapReduce under the hood

- ♦ Goal:
 - ♦ **Simple** programming model
 - ♦ **Batch** processing
 - ♦ High throughput
 - ♦ Often long running jobs
 - ♦ Not: low latency
 - ♦ **Scalability**
 - ♦ 10s, 100s, 1000s of nodes
 - ♦ Additionally depends on the problem
 - ♦ **Fault tolerance**
 - ♦ Cluster of commodity nodes
 - ♦ Handle failure of a node while job is running gracefully (query fault tolerance)

MapReduce execution overview

MapReduce execution overview

- ♦ Data (often) on distributed file system like GFS, HDFS

MapReduce execution overview

- ◆ Data (often) on distributed file system like GFS, HDFS
- ◆ One master, many workers
 - ◆ Input data **split** into M **map tasks** (typically 64 MB≈chunk size in GFS)
 - ◆ Reduce phase **partitioned** into R **reduce tasks** ($\text{hash}(\text{key}) \bmod R$)
 - ◆ Tasks are assigned to workers dynamically

MapReduce execution overview

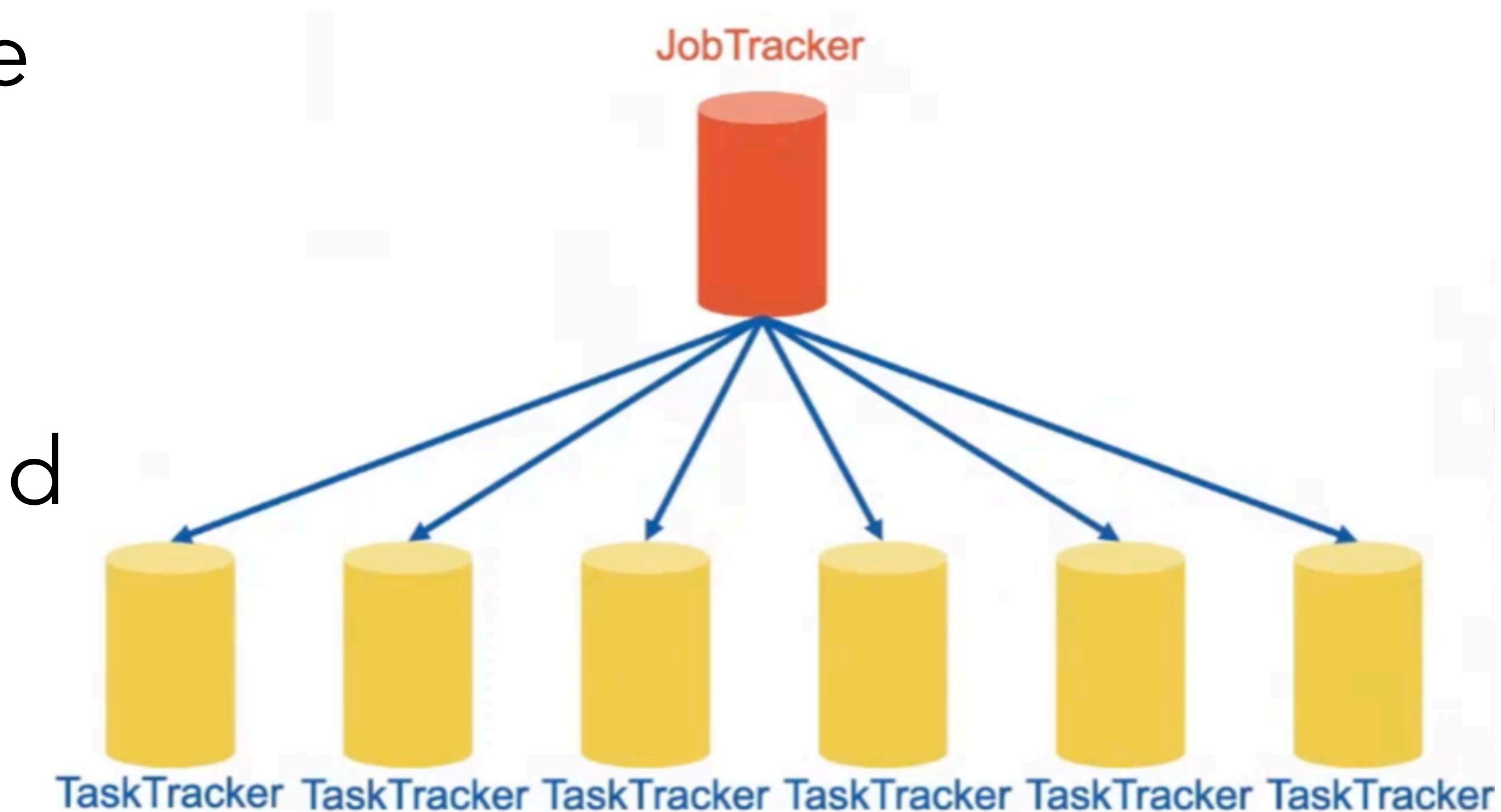
- ◆ Data (often) on distributed file system like GFS, HDFS
- ◆ One master, many workers
 - ◆ Input data **split** into M **map tasks** (typically 64 MB≈chunk size in GFS)
 - ◆ Reduce phase **partitioned** into R **reduce tasks** ($\text{hash}(\text{key}) \bmod R$)
 - ◆ Tasks are assigned to workers dynamically
- ◆ Master assigns each map task to a free worker
 - ◆ Considers **data locality** to worker when assigning a task
 - ◆ Worker reads task input (often from local files) and applies user's Map operation to each key-value pair
 - ◆ Worker produces R **local files** containing intermediate key-value pairs

MapReduce execution overview

- ◆ Data (often) on distributed file system like GFS, HDFS
- ◆ One master, many workers
 - ◆ Input data **split** into M **map tasks** (typically 64 MB≈chunk size in GFS)
 - ◆ Reduce phase **partitioned** into R **reduce tasks** ($\text{hash}(\text{key}) \bmod R$)
 - ◆ Tasks are assigned to workers dynamically
- ◆ Master assigns each map task to a free worker
 - ◆ Considers **data locality** to worker when assigning a task
 - ◆ Worker reads task input (often from local files) and applies user's Map operation to each key-value pair
 - ◆ Worker produces R **local files** containing intermediate key-value pairs
- ◆ Master assigns each reduce task to a free worker
 - ◆ Worker reads intermediate key-value pairs from map workers
 - ◆ Worker sorts and applies user's Reduce operation to produce the output in global filesystem

Hadoop architecture

- ◆ **Jobtracker:** client communication, job scheduling, resource management, lifecycle coordination
- ◆ **Tasktracker:** task execution and management



Data locality

Data locality

- ♦ Move **computation to data**
 - ♦ Small code, large data
 - ♦ Goal: Reduce network bandwidth

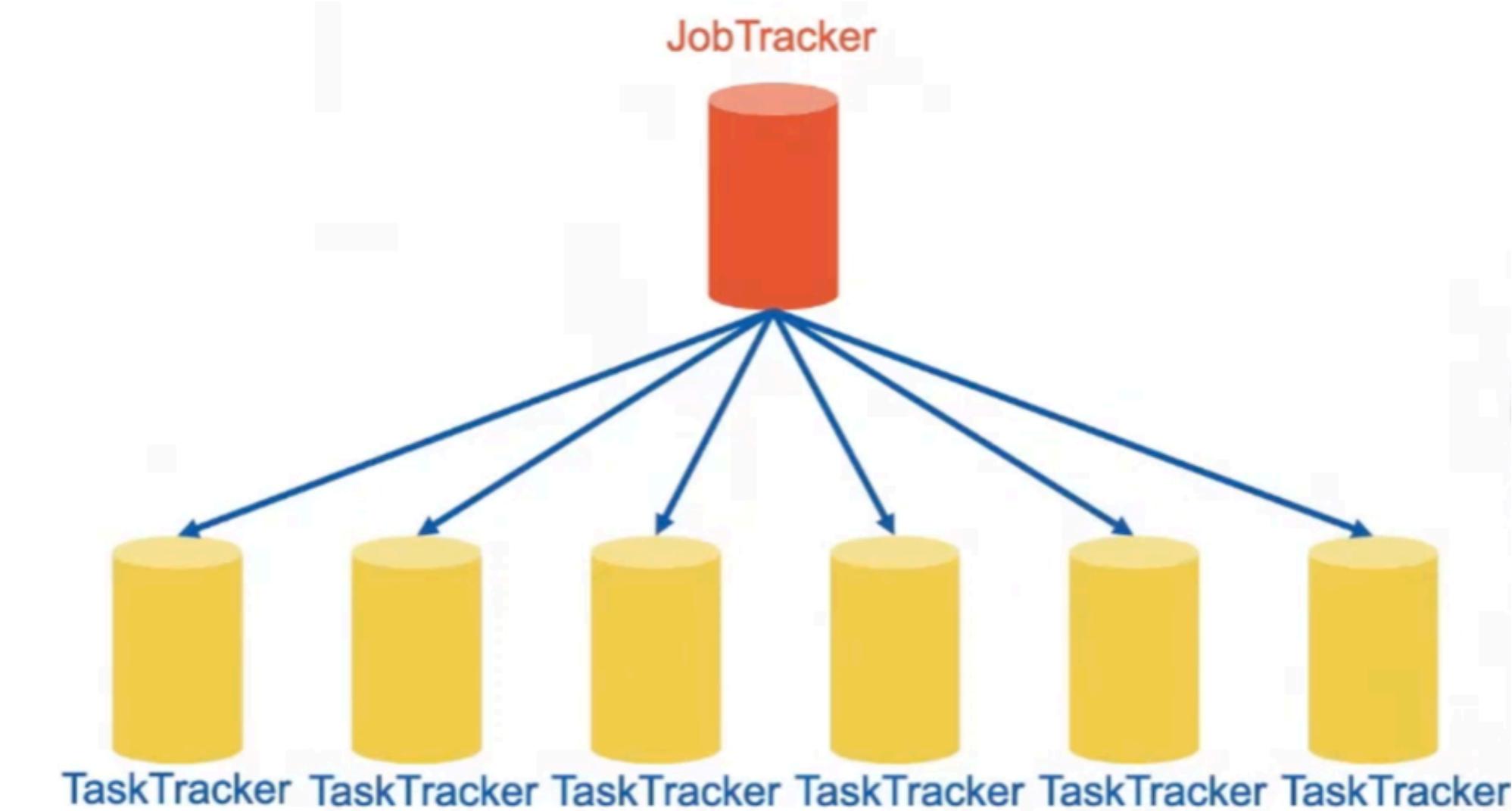
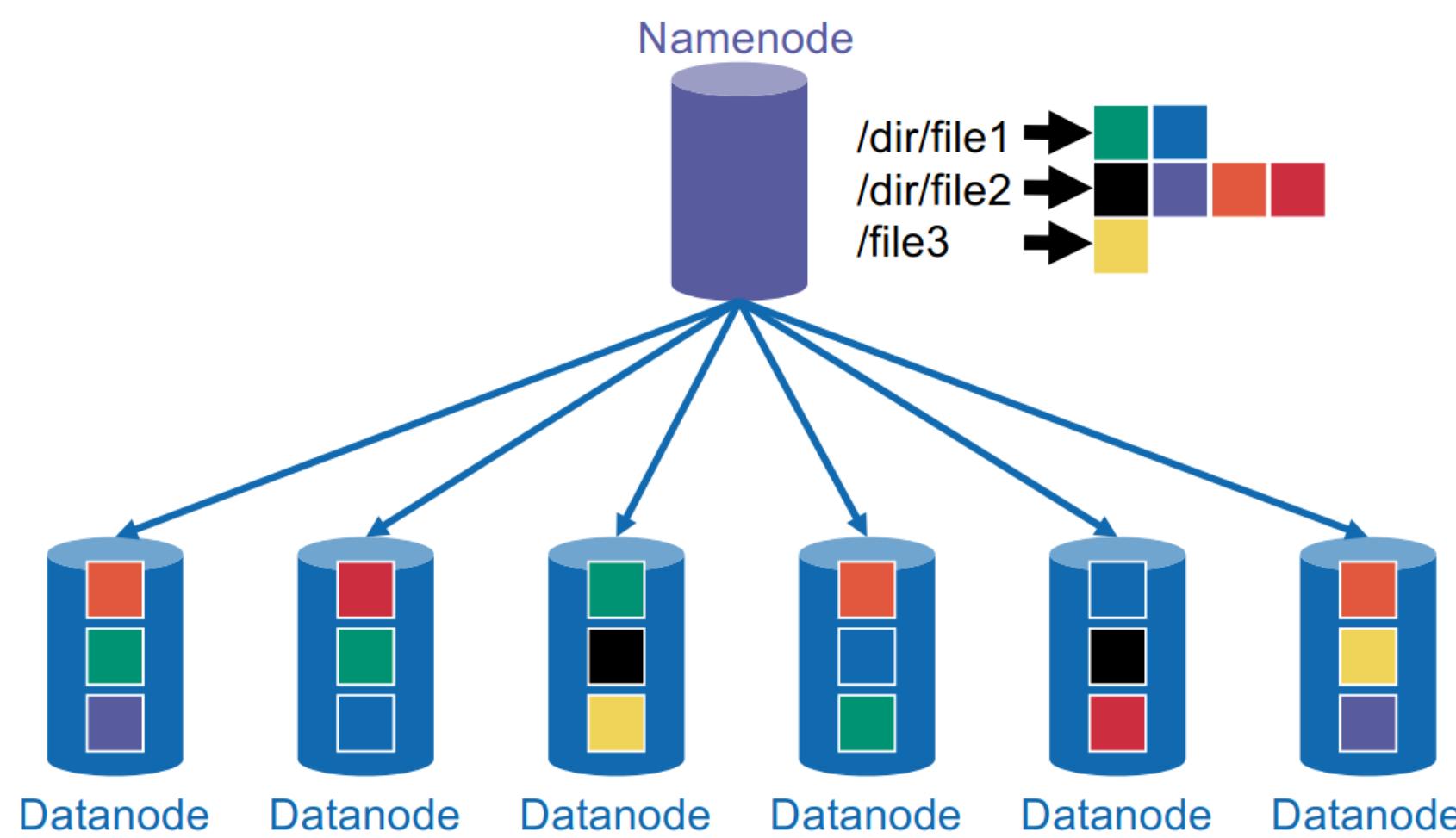
Data locality

- ◆ Move **computation to data**
 - ◆ Small code, large data
 - ◆ Goal: Reduce network bandwidth
- ◆ Machines run both chunkservers/datanodes and MR workers
 - ◆ Each chunk stored at 3 machines
 - ◆ MR master **schedules map tasks based on the location of these chunks**

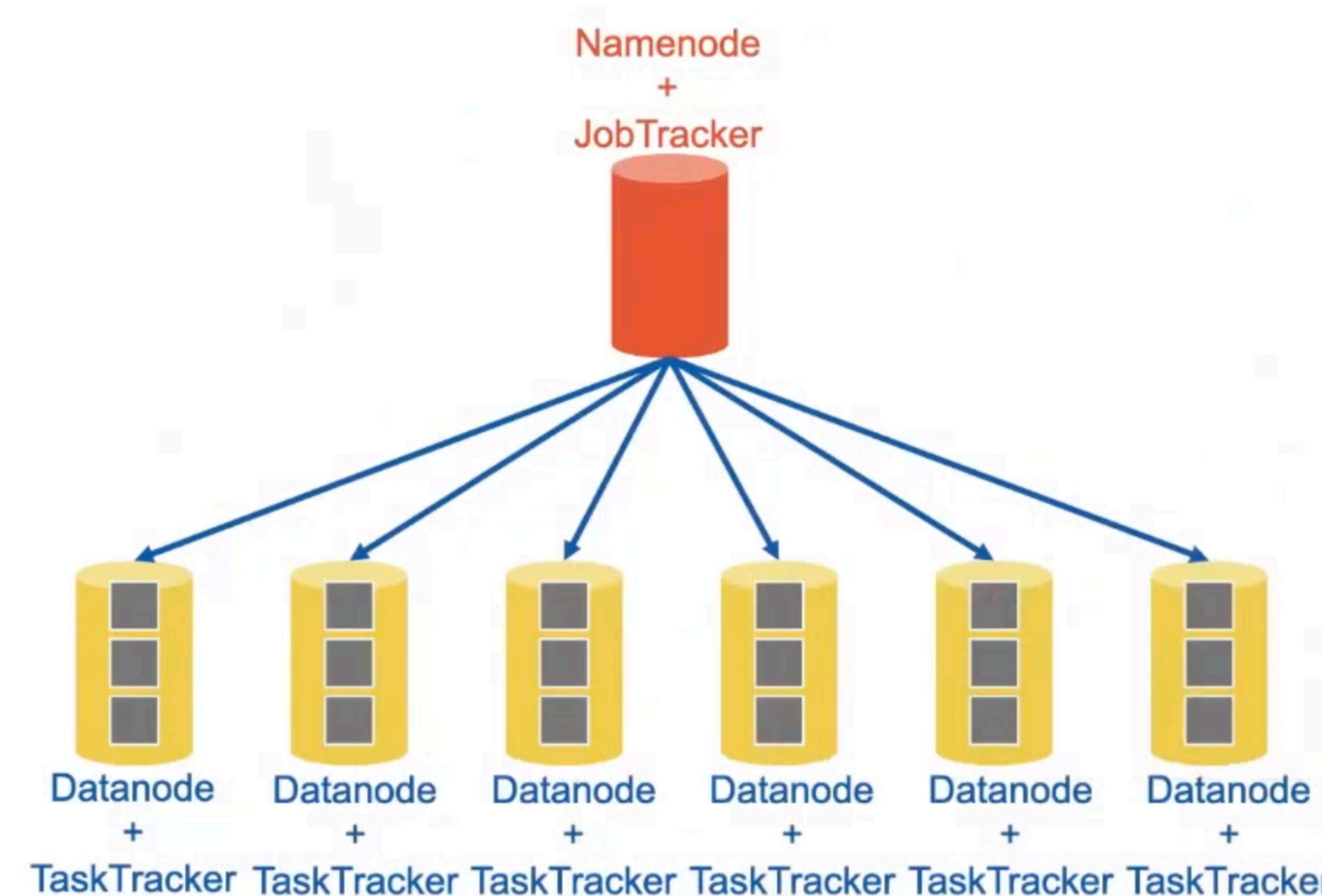
Data locality

- ◆ Move **computation to data**
 - ◆ Small code, large data
 - ◆ Goal: Reduce network bandwidth
- ◆ Machines run both chunkservers/datanodes and MR workers
 - ◆ Each chunk stored at 3 machines
 - ◆ MR master **schedules map tasks based on the location of these chunks**
- ◆ Goal: run map task on machine that stores input chunk
 - ◆ Then map task reads data locally (**local fetch**)
 - ◆ Thousands of machines can read at local disk speed; read rate not limited by network

Hadoop infrastructure (first version)



Hadoop infrastructure (first version)



SQL in MapReduce

- ◆ MapReduce is powerful enough to run basic SQL queries
 - ◆ Although that's not its original intention
- ◆ Easy case: single table, no sorting
 - ◆

```
select prodId, sum(amt)
      from sales
     where depId = 10
  groupby prodId
```
 - ◆ Map: selection, projection w/o duplicate elimination
 - ◆ MapReduce framework: grouping
 - ◆ Reduce: aggregation, having, duplicate elimination

Joins in MapReduce

- ♦ Two main strategies:
 - ♦ **symmetric hash join (repartition join)**
 - ♦ **replicated join (broadcast join)**

Repartition join

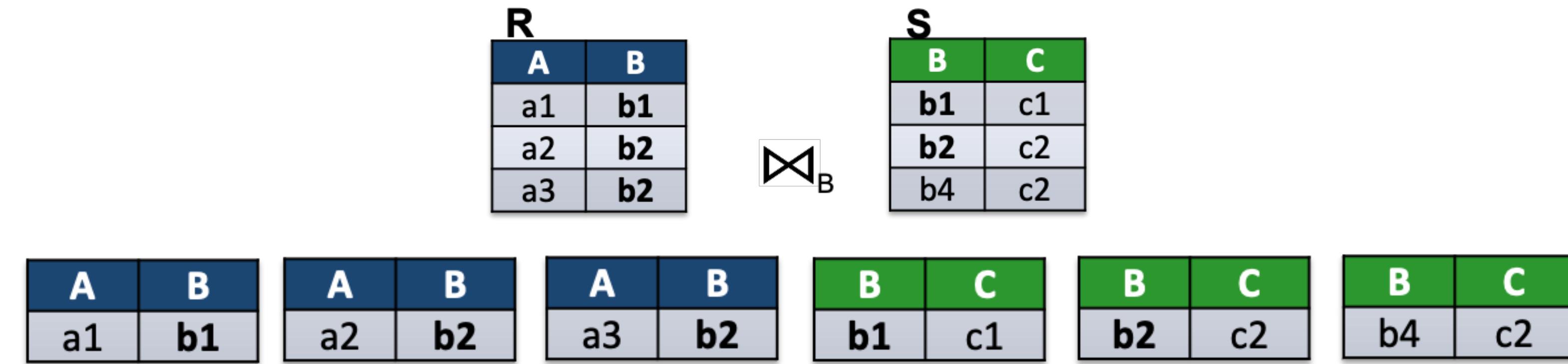
Repartition join

R	A	B
a1	b1	
a2	b2	
a3	b2	

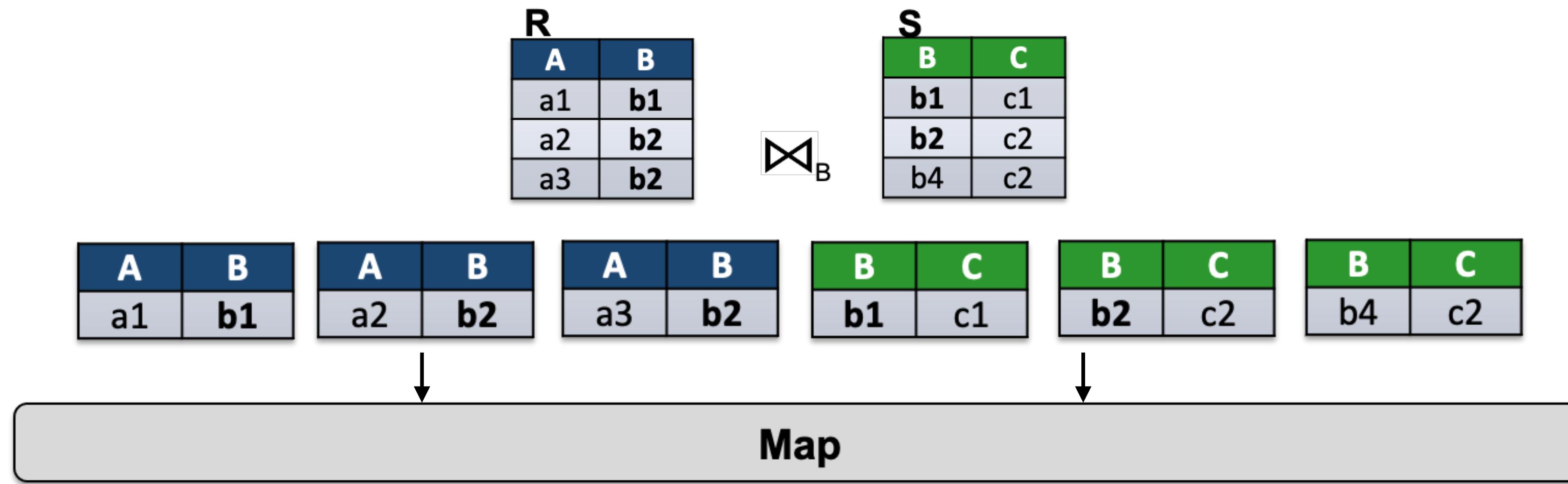
\bowtie_B

S	B	C
b1		c1
b2		c2
b4		c2

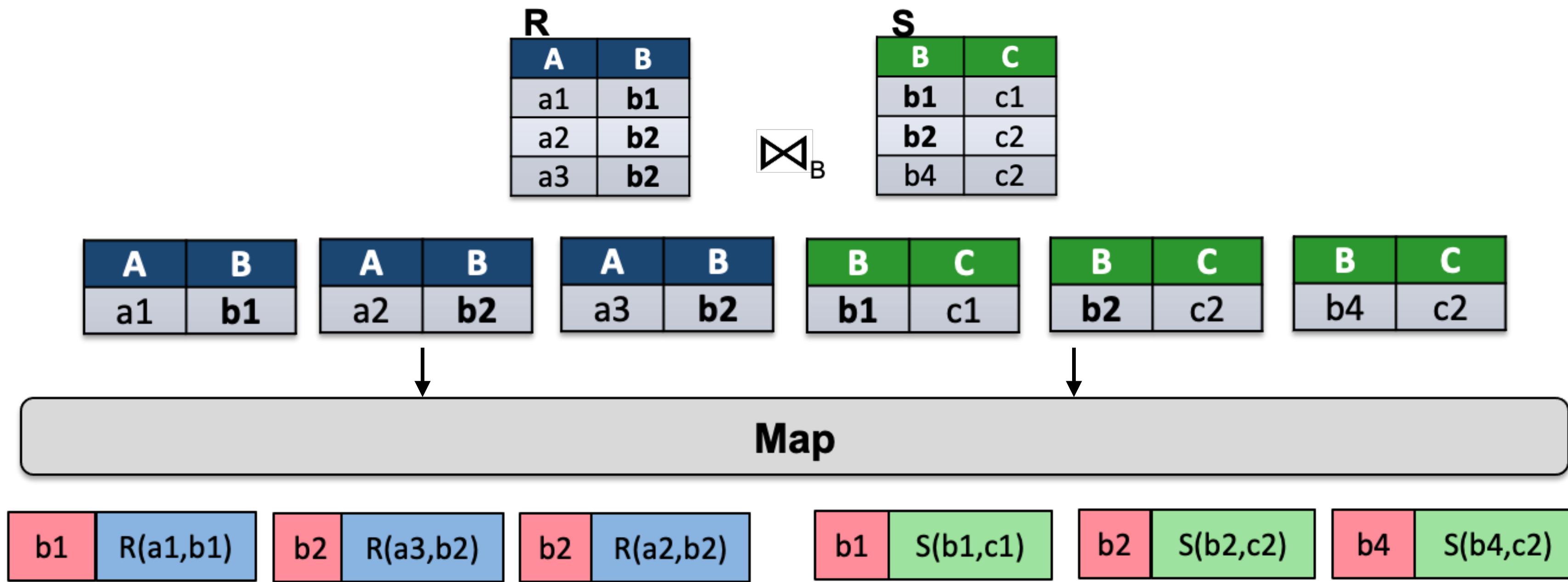
Repartition join



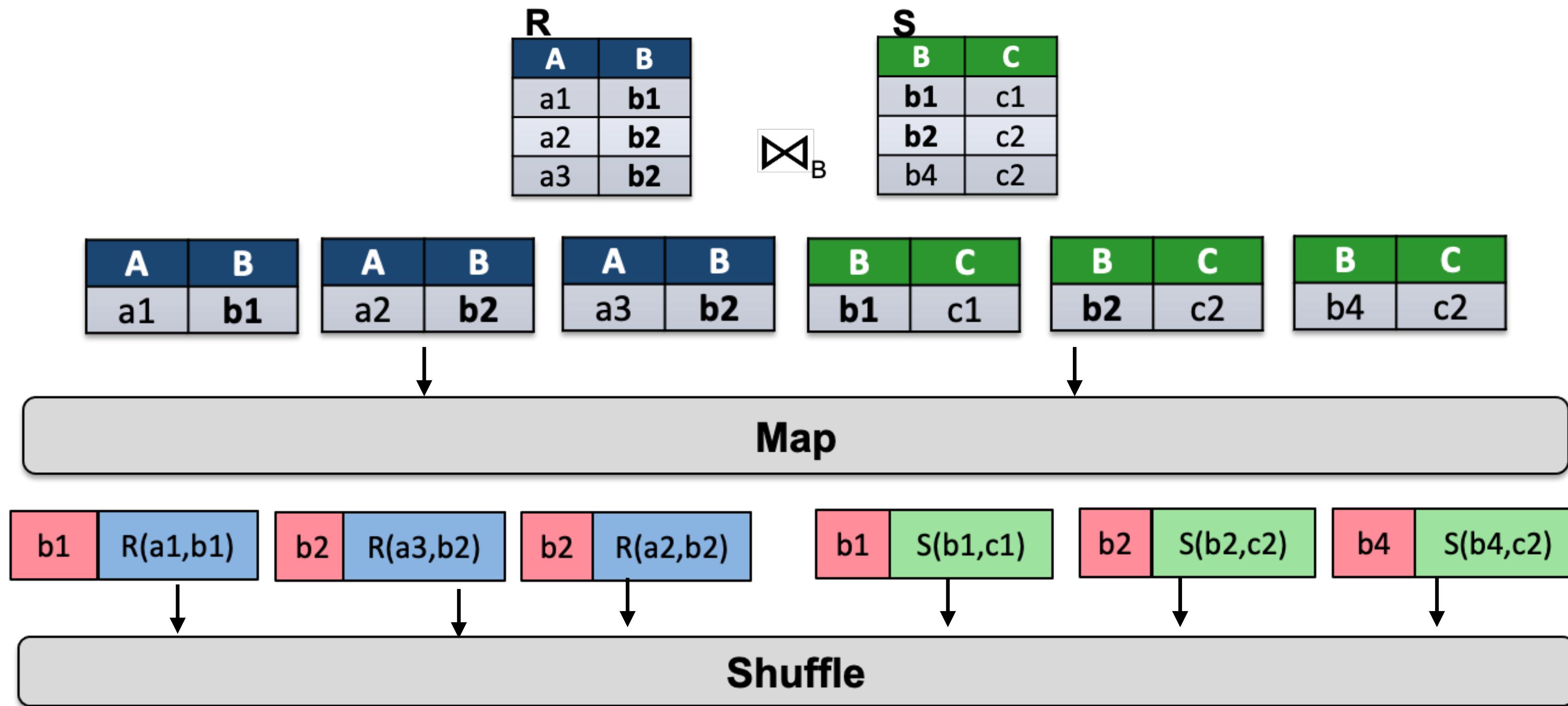
Repartition join



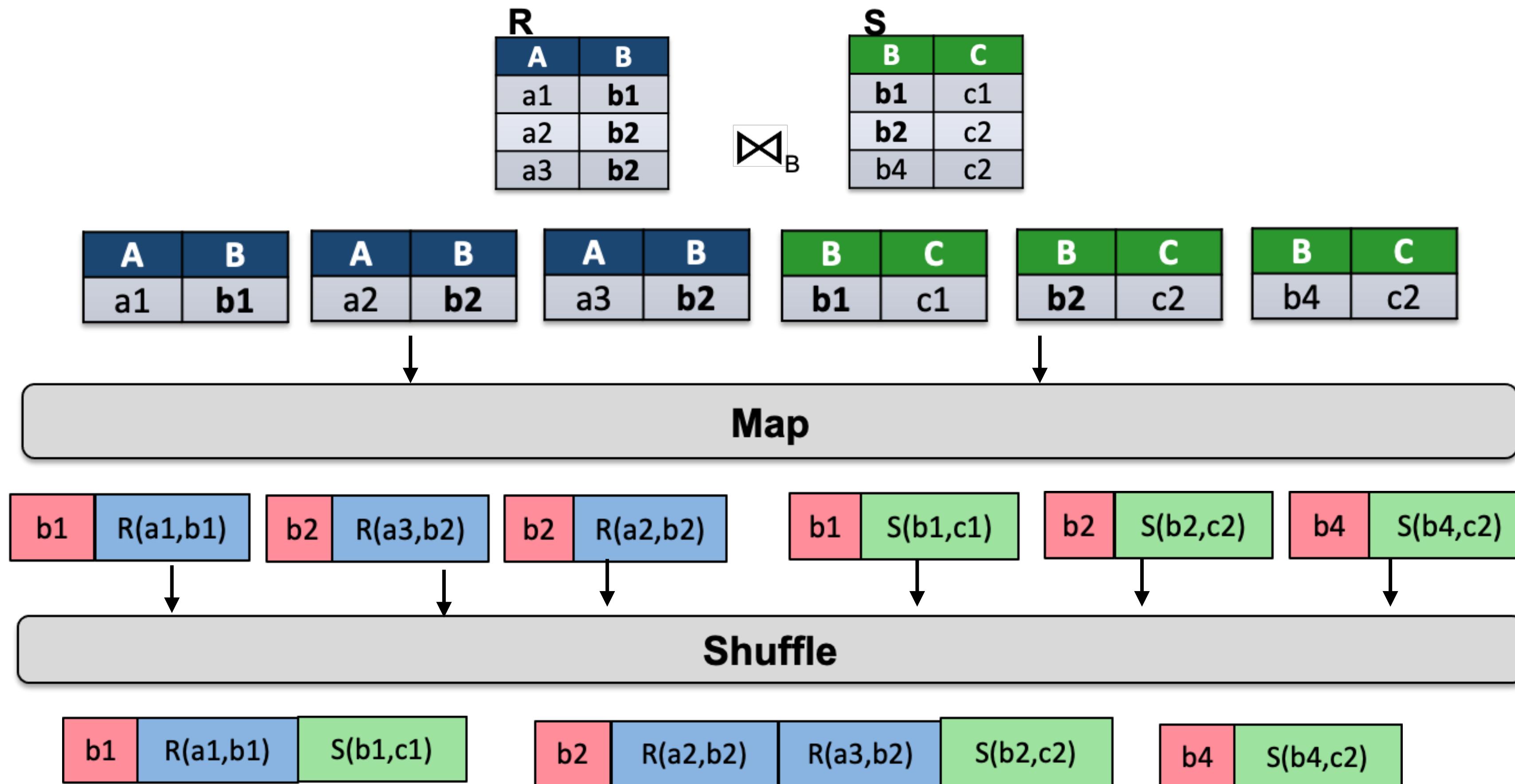
Repartition join



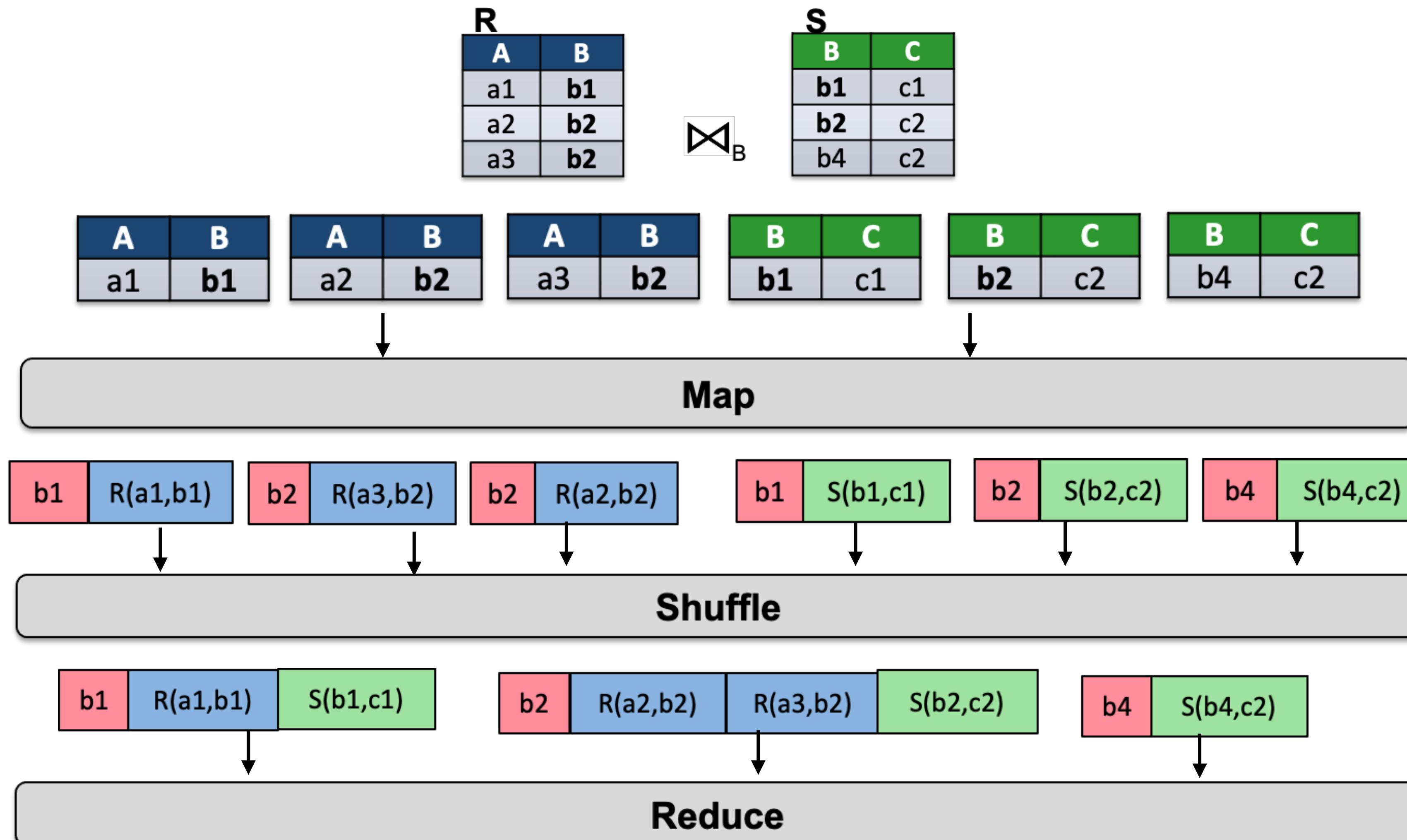
Repartition join



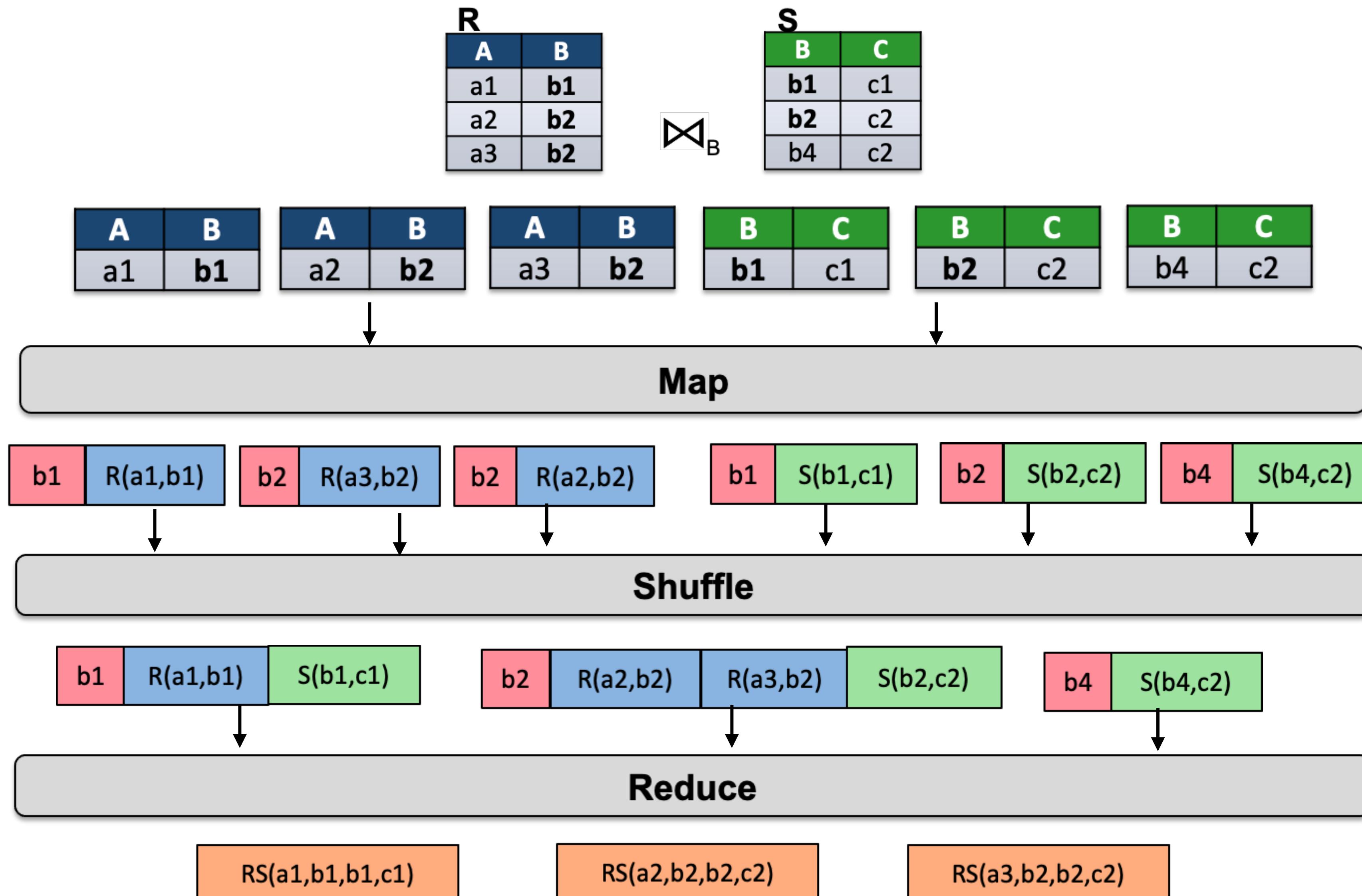
Repartition join



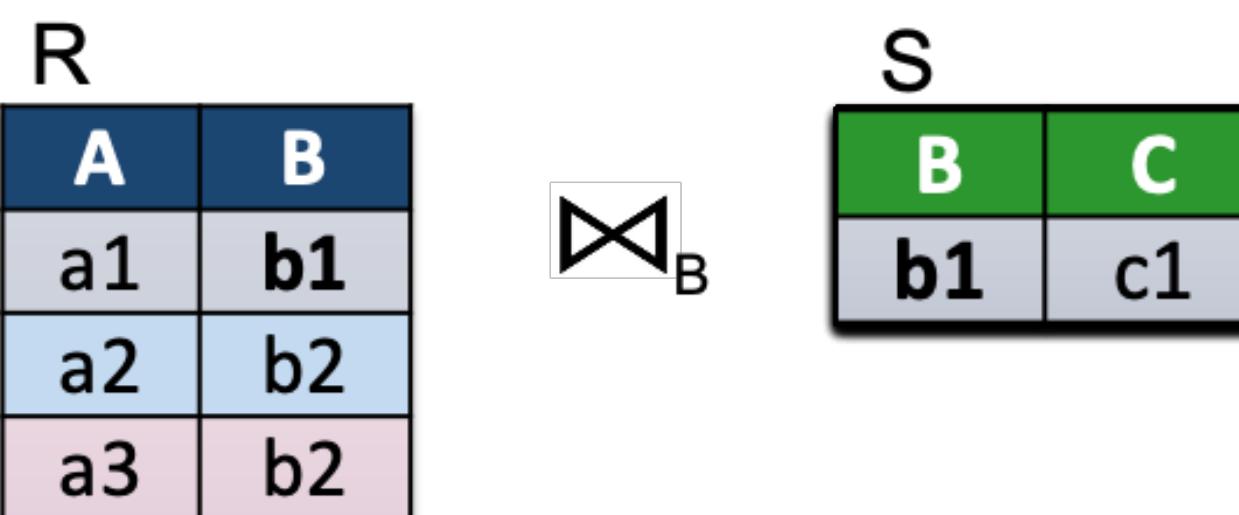
Repartition join



Repartition join



Broadcast join



Broadcast join

R	A	B
a1	b1	
a2	b2	
a3	b2	



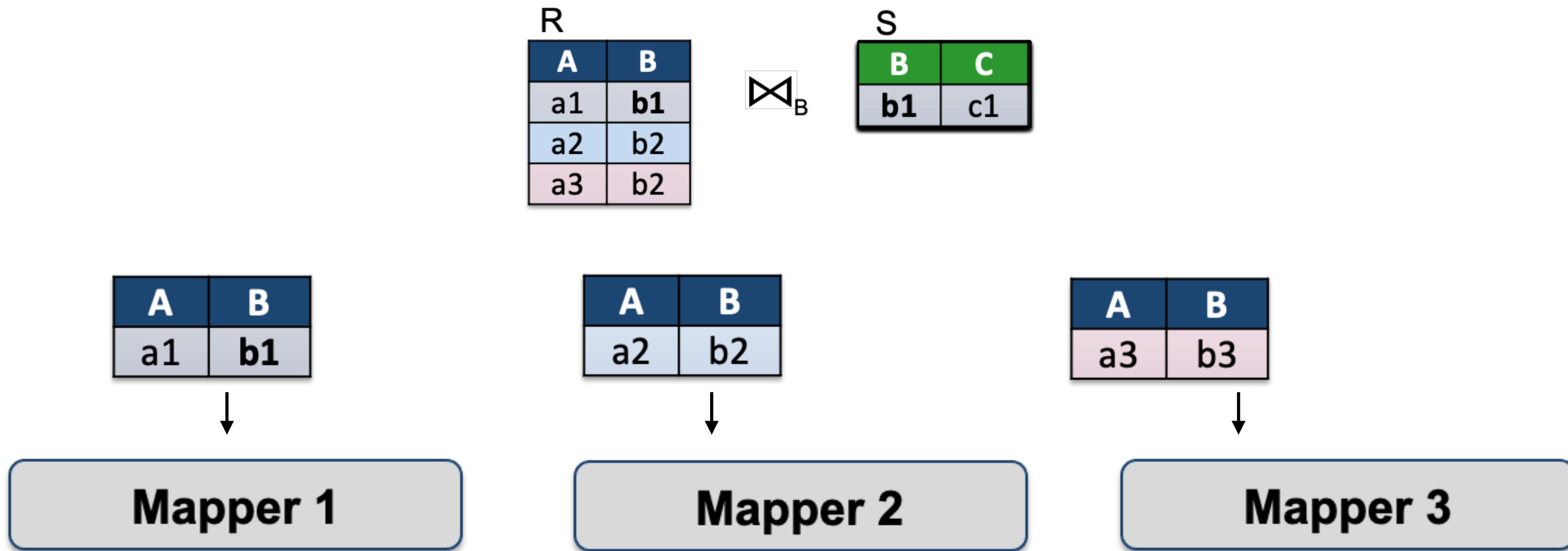
S	B	C
b1	c1	

A	B
a1	b1

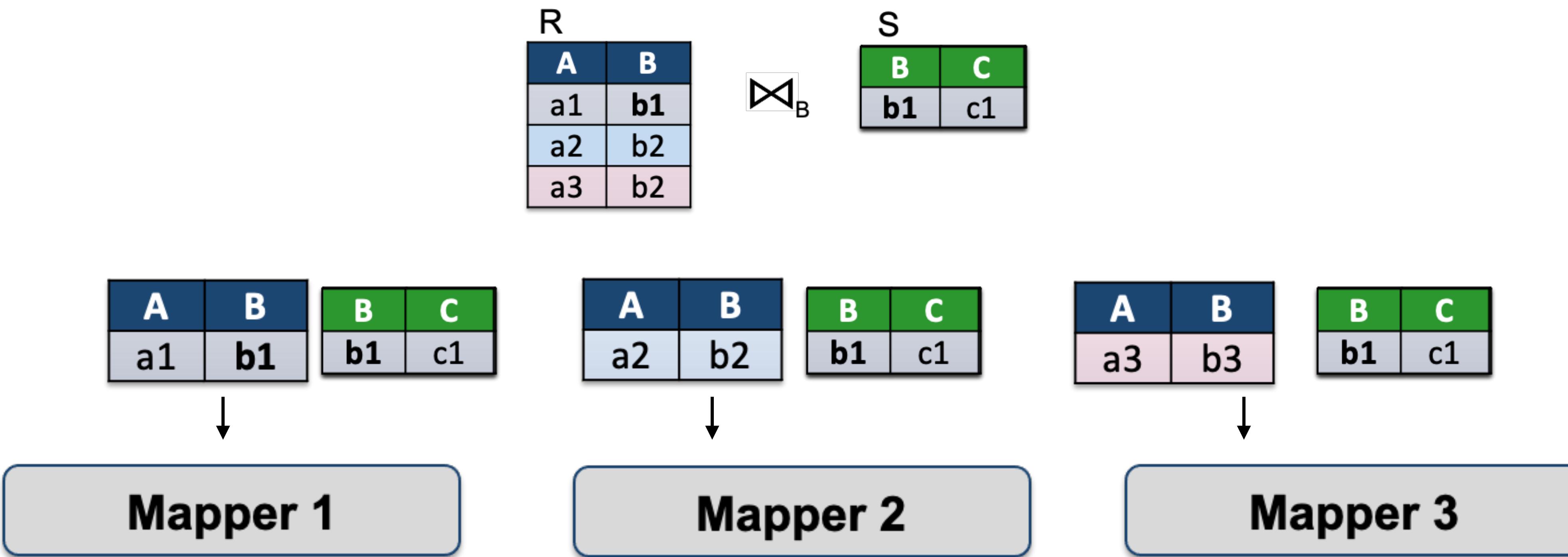
A	B
a2	b2

A	B
a3	b3

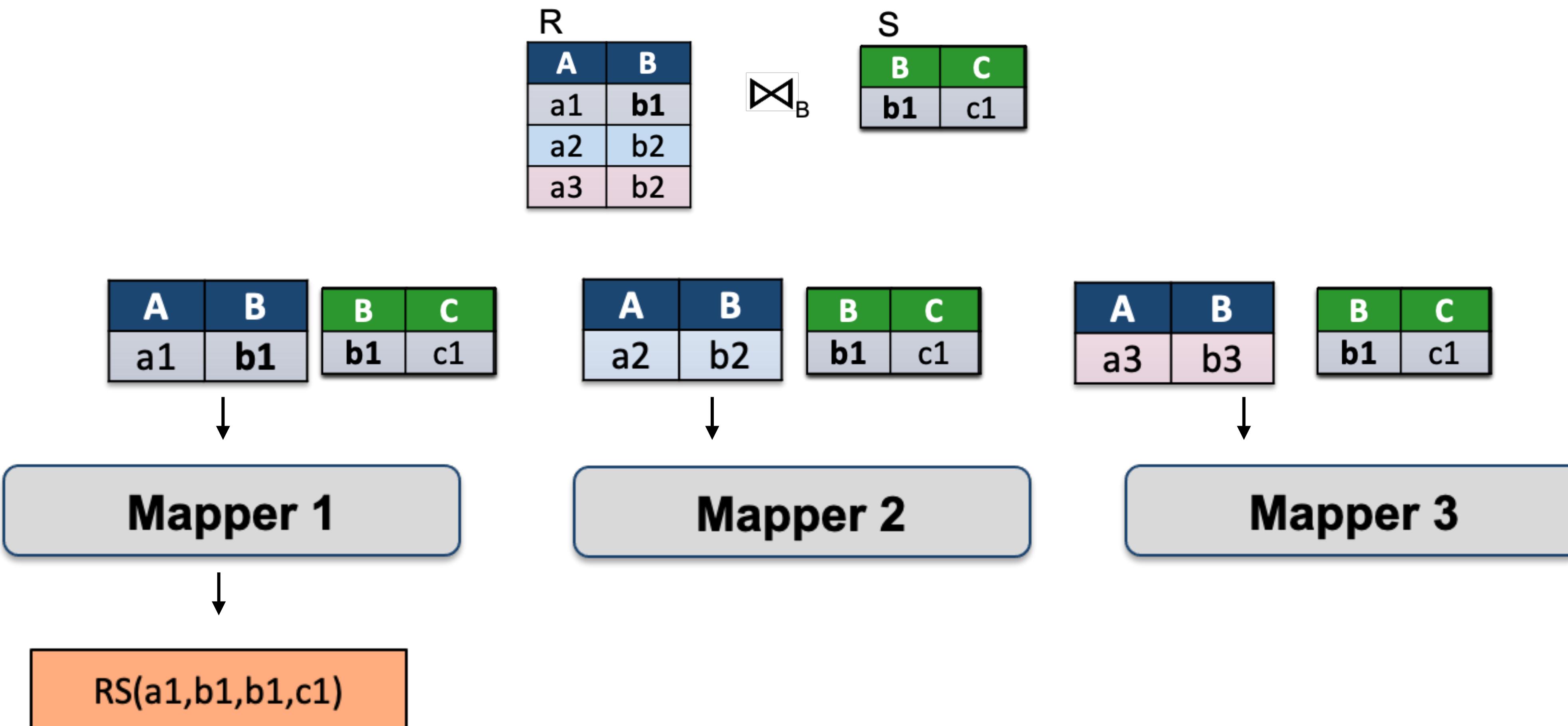
Broadcast join



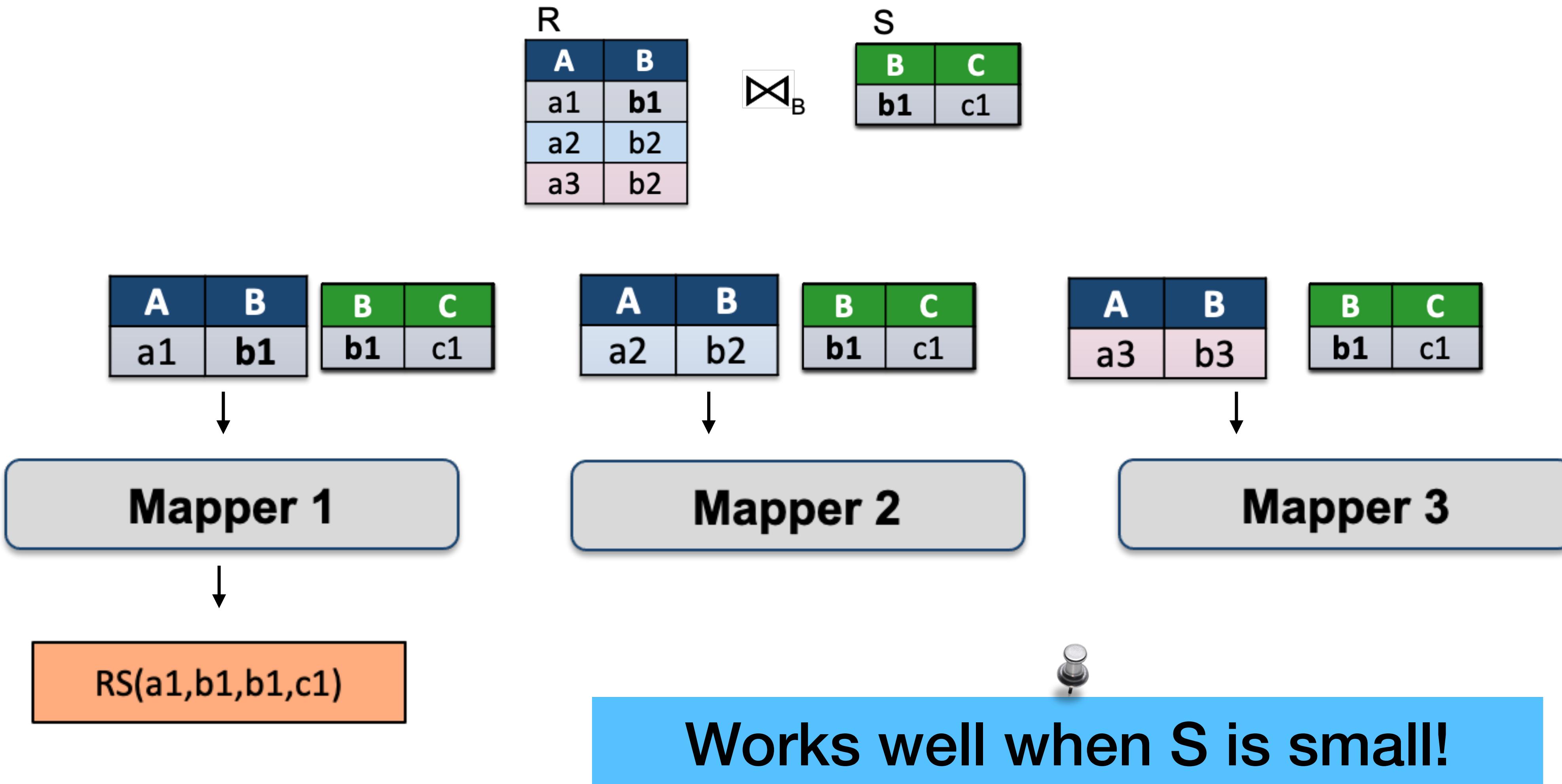
Broadcast join



Broadcast join



Broadcast join



Today's lecture

- ◆ Distributed file systems (HDFS)
- ◆ Distributed data processing paradigms (MapReduce, Hadoop)
- ◆ **Dataflow engines (Apache Spark)**



Problems with MapReduce

MapReduce: A major step backwards

By David DeWitt on January 17, 2008 4:20 PM | [Permalink](#) | [Comments \(44\)](#) | [TrackBacks \(1\)](#)
[Note: Although the system attributes this post to a single author, it was written by David J. DeWitt and Michael Stonebraker]

On January 8, a Database Column reader asked for our views on new distributed database research efforts, and we'll begin here with our views on [MapReduce](#). This is a good time to discuss it, since the recent trade press has been filled with news of the revolution of so-called "cloud computing." This paradigm entails harnessing large numbers of (low-end) processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a large number of "jelly beans" rather than utilizing a much smaller number of high-end servers.

For example, IBM and Google have announced plans to make a 1,000 processor cluster available to a few select universities to teach students how to program such clusters using a software tool called MapReduce [1]. Berkeley has gone so far as to plan on teaching their freshman how to program using the MapReduce framework.

As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represents a paradigm shift in the development of scalable, data-intensive applications. MapReduce may be a good idea for writing certain types of general-purpose computations, but to the database community, it is:

1. A giant step backward in the programming paradigm for large-scale data intensive applications
2. A sub-optimal implementation, in that it uses brute force instead of indexing
3. Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago
4. Missing most of the features that are routinely included in current DBMS
5. Incompatible with all of the tools DBMS users have come to depend on

How to improve MapReduce?

- ♦ Direction 1: **Higher-level languages**
 - ♦ Write programs in a higher-level language
 - ♦ Programs are “transformed into” MapReduce job(s)
 - ♦ Addresses (mainly) **programmability**

How to improve MapReduce?

♦ Direction 1: Higher-level languages

- ♦ Write programs in a higher-level language
- ♦ Programs are “transformed into” MapReduce job(s)
- ♦ Addresses (mainly) **programmability**

Examples

Hive
(Facebook)



```
SELECT count (*)  
FROM users
```

Pig Latin
(Yahoo)



```
A = load 'users';  
B = group A all;  
C = foreach B generate COUNT(A)
```

How to improve MapReduce?

- ♦ Direction 2: **Dataflow engines**
 - ♦ Instead of compiling to MapReduce, build a more suitable execution environment
 - ♦ Addresses **performance**

How to improve MapReduce?

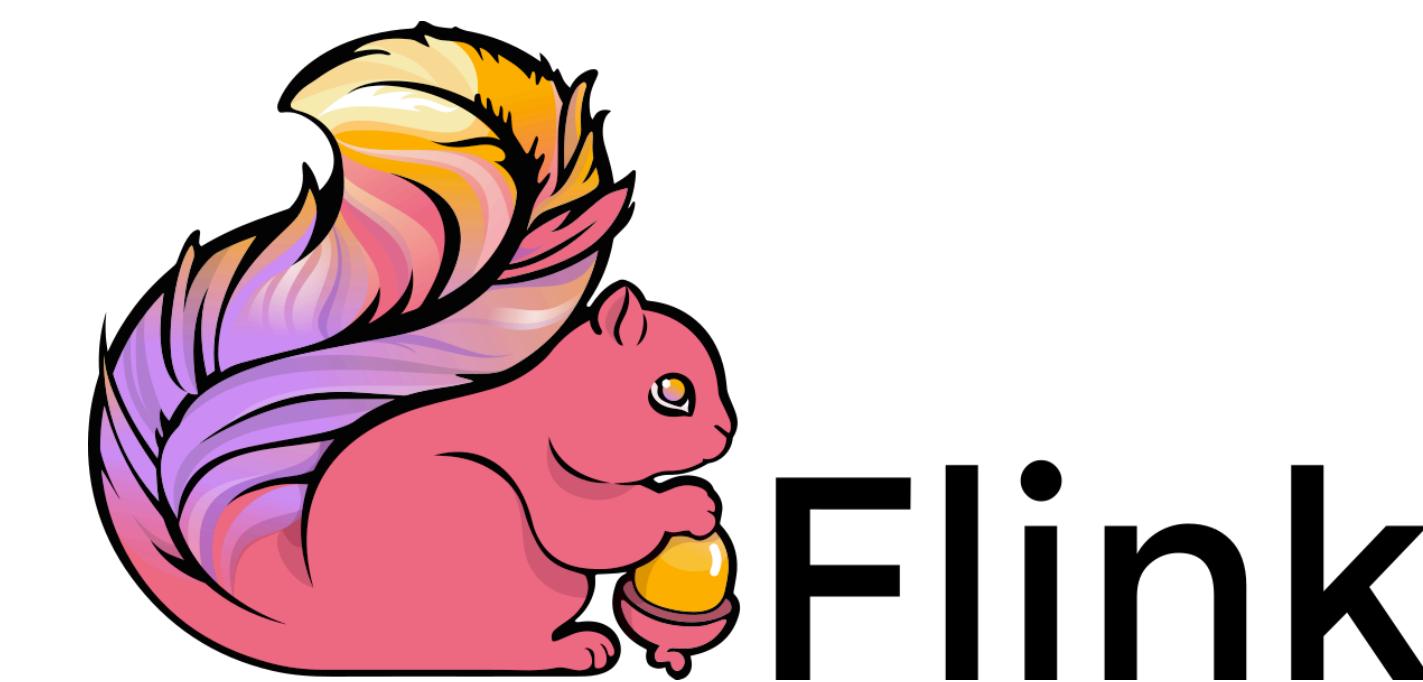
- ♦ Direction 2: Dataflow engines
- ♦ Instead of compiling to MapReduce, build a more suitable execution environment
- ♦ Addresses performance

Examples

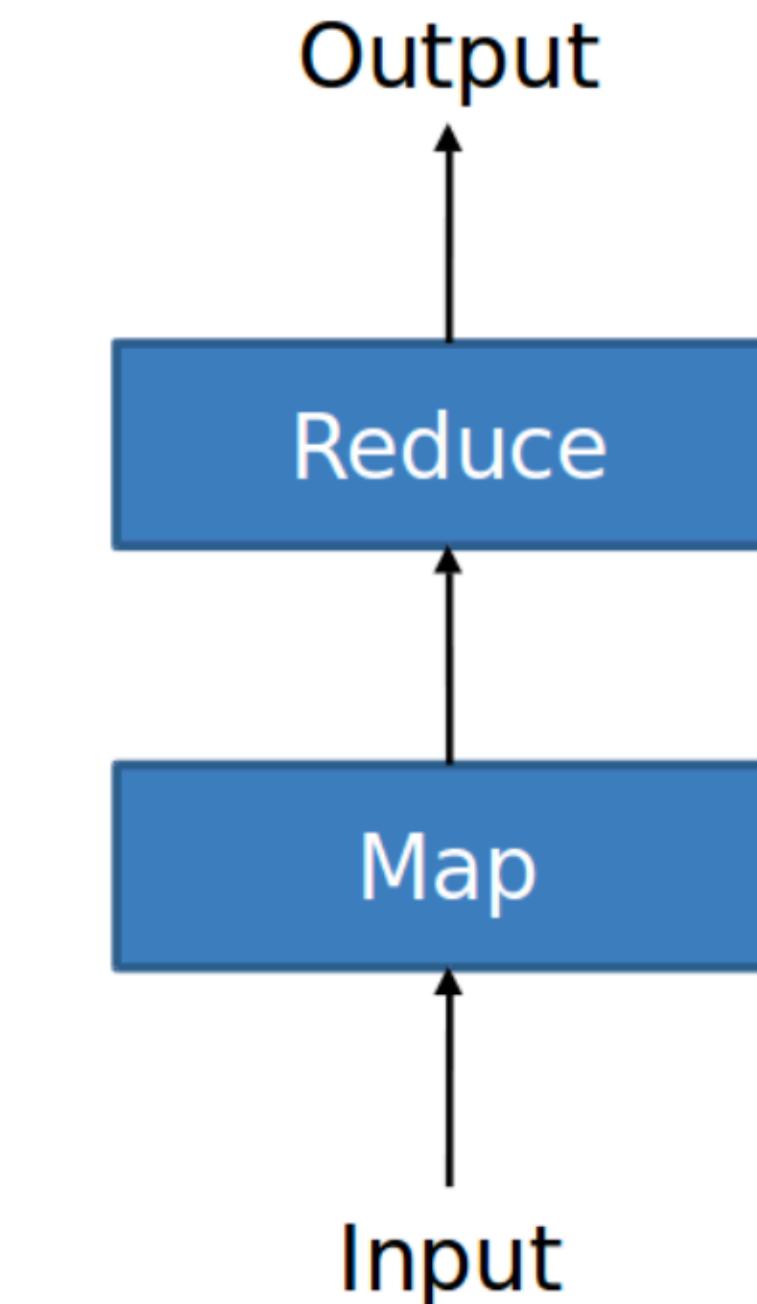
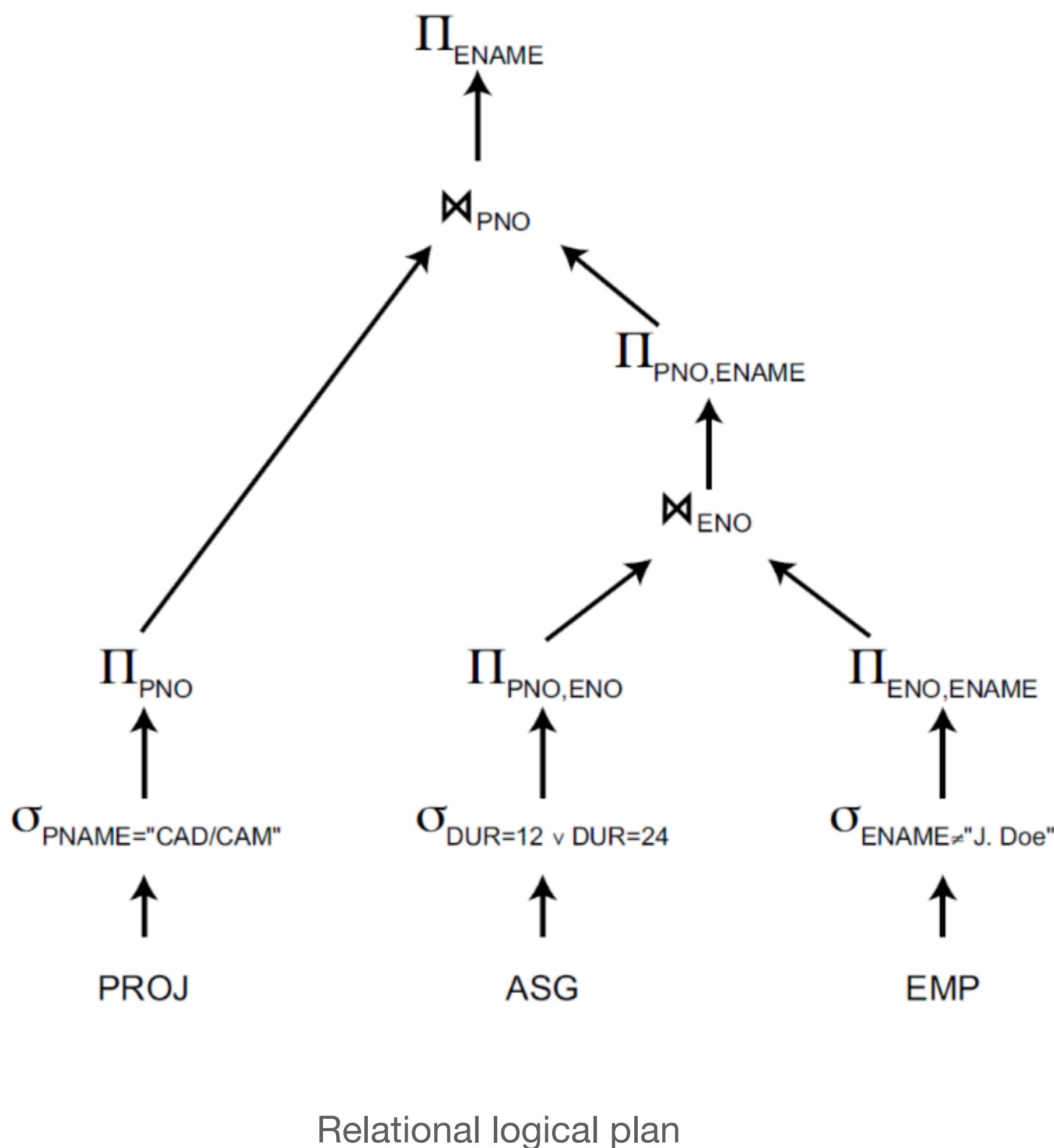
Apache Spark
(UC Berkeley)



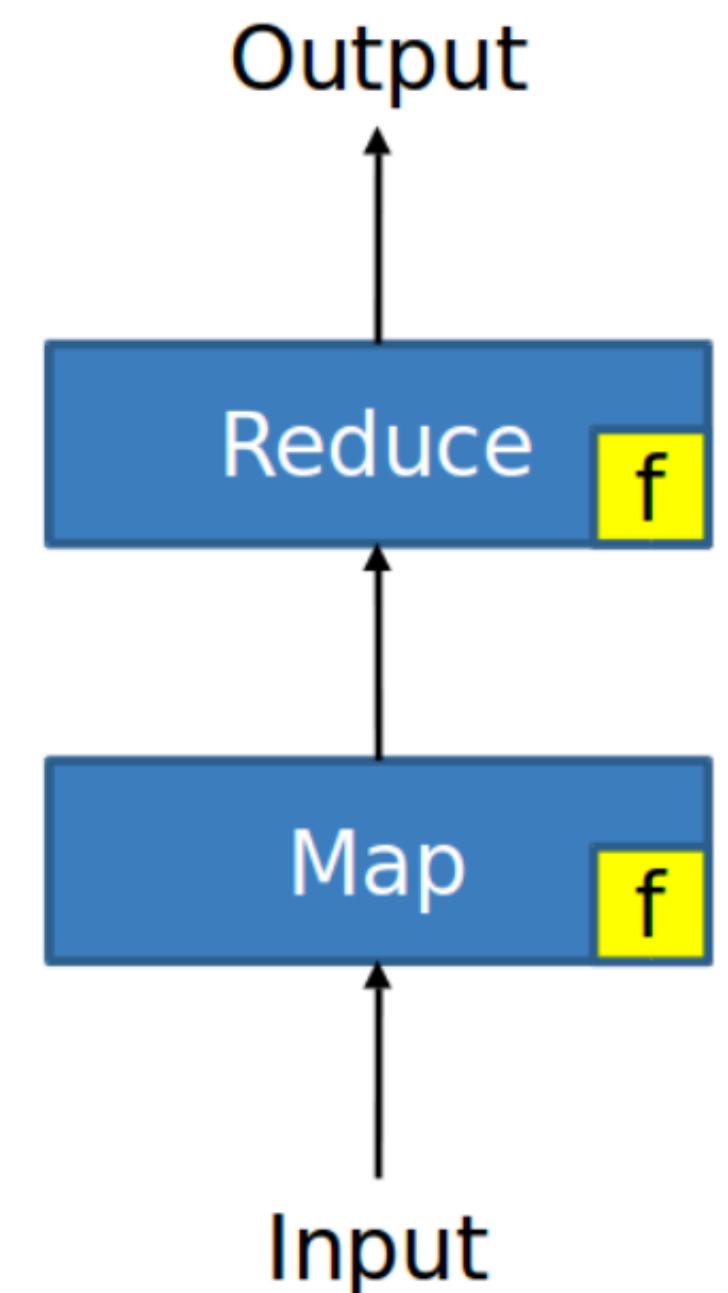
Apache Flink
(TU Berlin)



Example (logical) dataflow

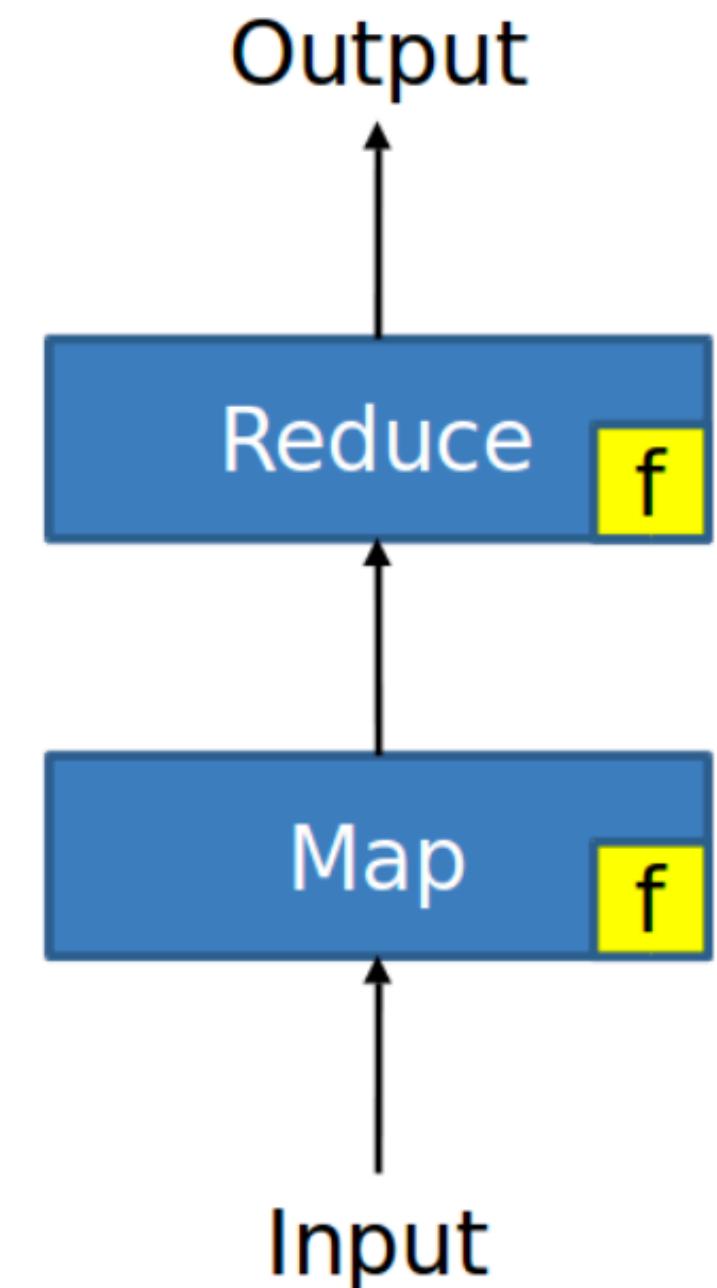


MapReduce as dataflow



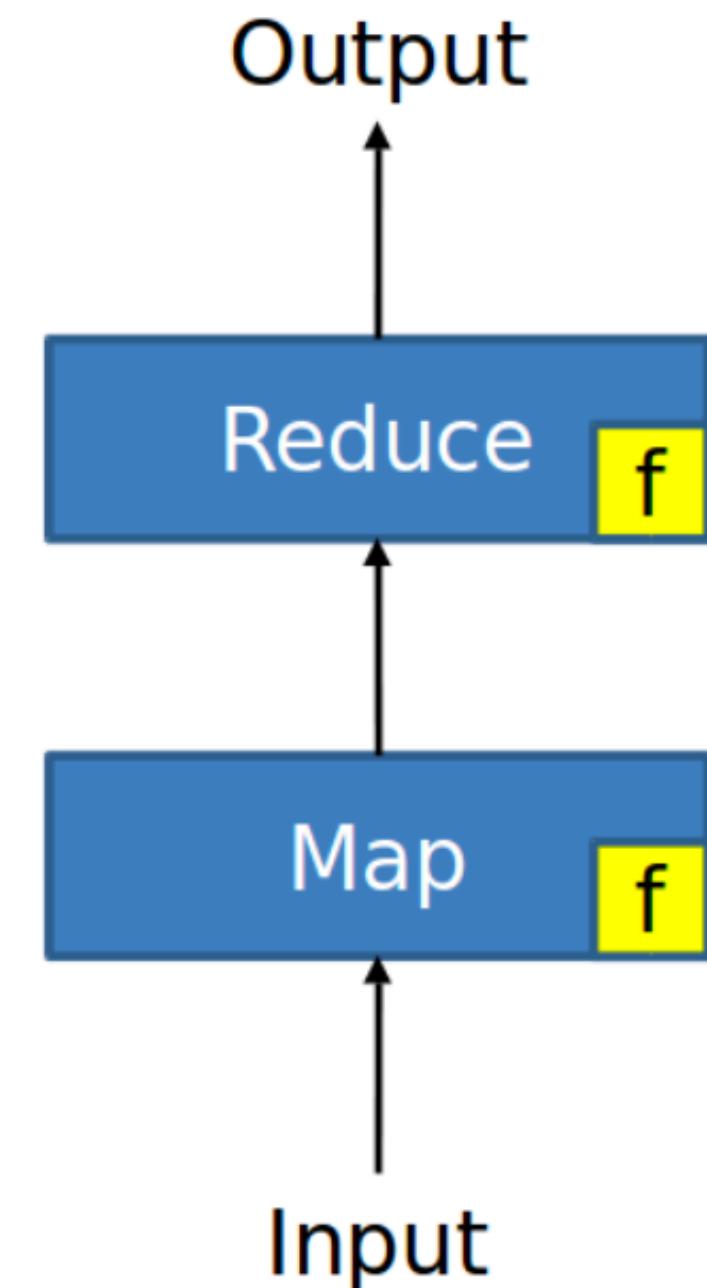
MapReduce as dataflow

- ♦ Two operators, fixed logical dataflow
 - ♦ Key: operators are **parameterized by user defined functions** (UDF)



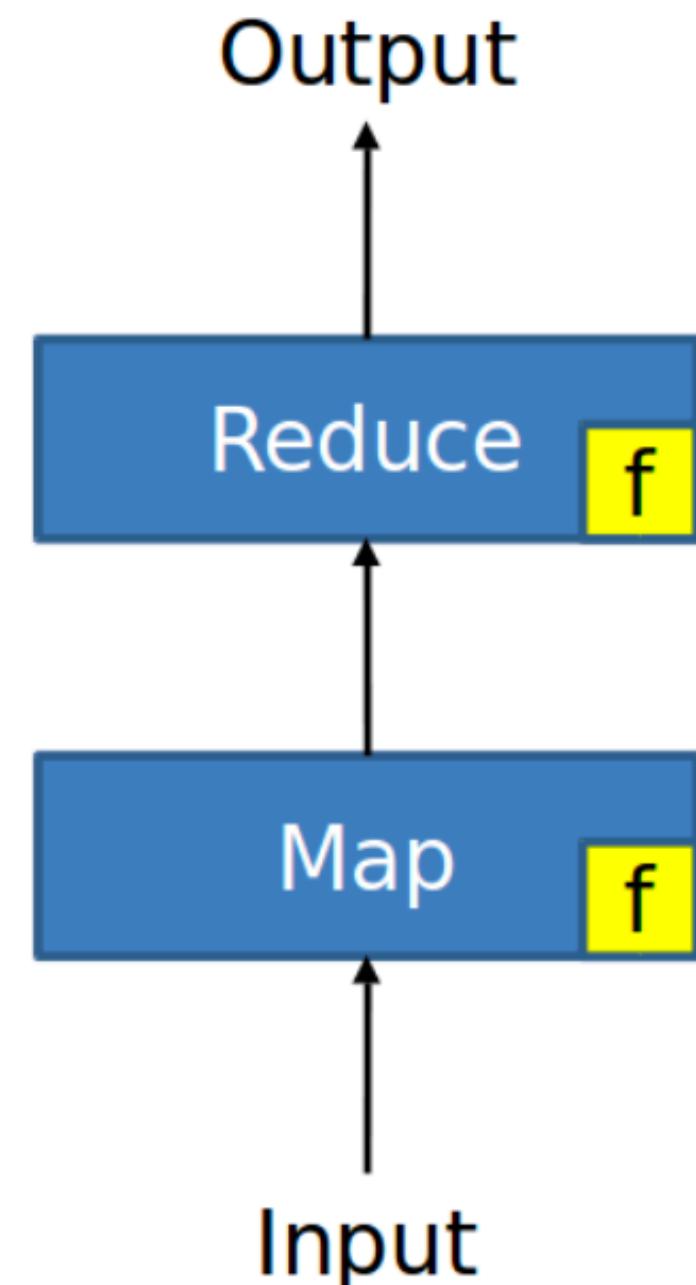
MapReduce as dataflow

- ◆ Two operators, fixed logical dataflow
 - ◆ Key: operators are **parameterized by user defined functions** (UDF)
- ◆ Automatic parallelization at runtime
 - ◆ Executes one of the possible physical dataflows (#maps, #reduces,...)



MapReduce as dataflow

- ◆ Two operators, fixed logical dataflow
 - ◆ Key: operators are **parameterized by user defined functions** (UDF)
- ◆ Automatic parallelization at runtime
 - ◆ Executes one of the possible physical dataflows (#maps, #reduces,...)
- ◆ **Dataflow** engines push this idea further
 - ◆ Keep UDFs
 - ◆ Add more operators
 - ◆ Improve implementation
 - ◆ Add logical/physical optimizations



Core abstraction: RDD

Core abstraction: RDD

- ♦ Resilient distributed dataset (RDD)
 - ♦ Read-only, partitioned collections of records (conceptually)
 - ♦ Spread across a cluster, stored in RAM, on disk, ...
 - ♦ Built through parallel **coarse-grained** transformations
 - ♦ Automatically rebuilt on failure (via lineage)

Core abstraction: RDD

- ♦ Resilient distributed dataset (RDD)
 - ♦ Read-only, partitioned collections of records (conceptually)
 - ♦ Spread across a cluster, stored in RAM, on disk, ...
 - ♦ Built through parallel **coarse-grained** transformations
 - ♦ Automatically rebuilt on failure (via lineage)
- ♦ Write programs in terms of distributed datasets and operations on them

Working with RDDs (I)

3

Working with RDDs (I)

```
lines = sc.textFile("hdfs://data.txt")
```

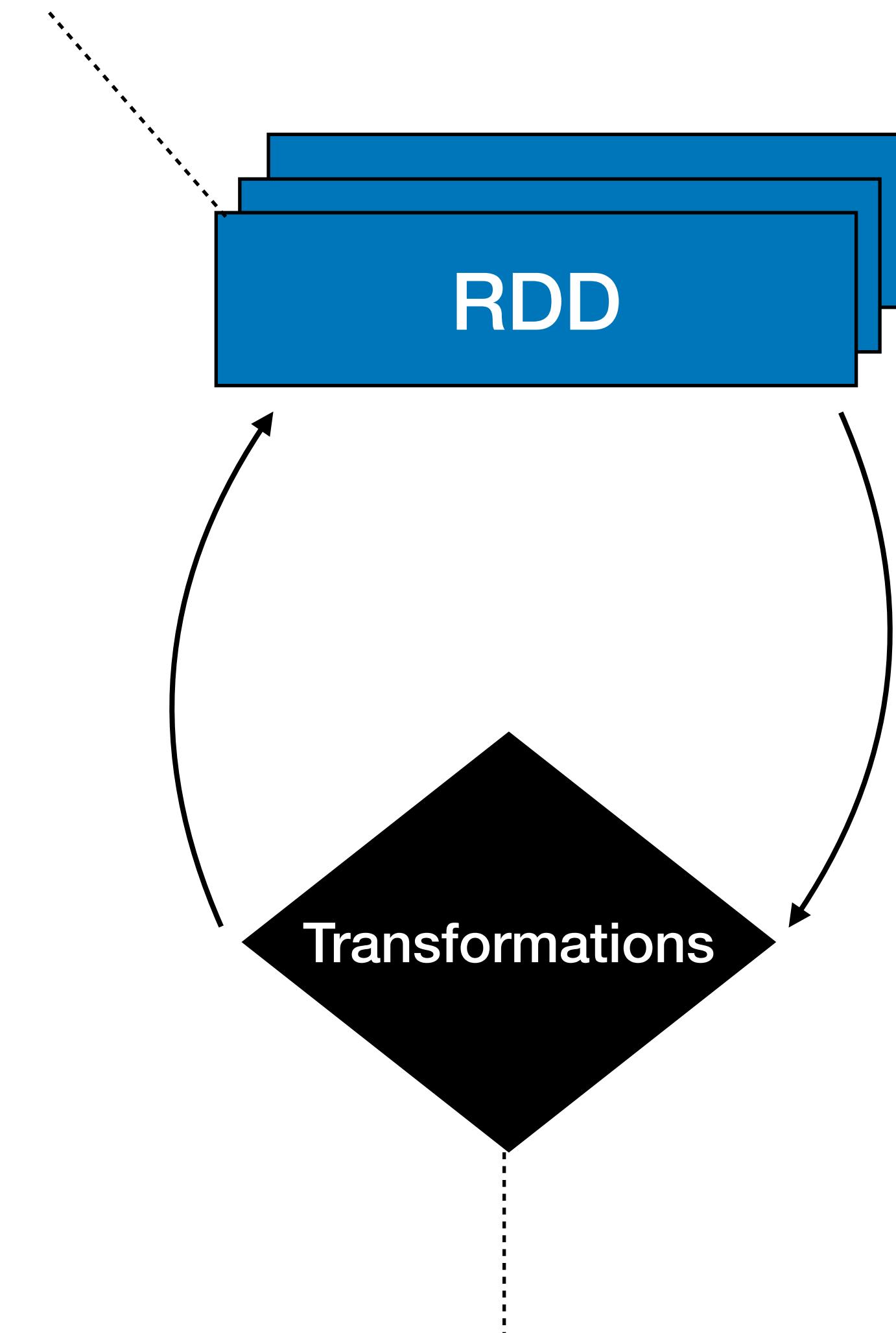


RDD

3

Working with RDDs (I)

```
lines = sc.textFile("hdfs://data.txt")
```

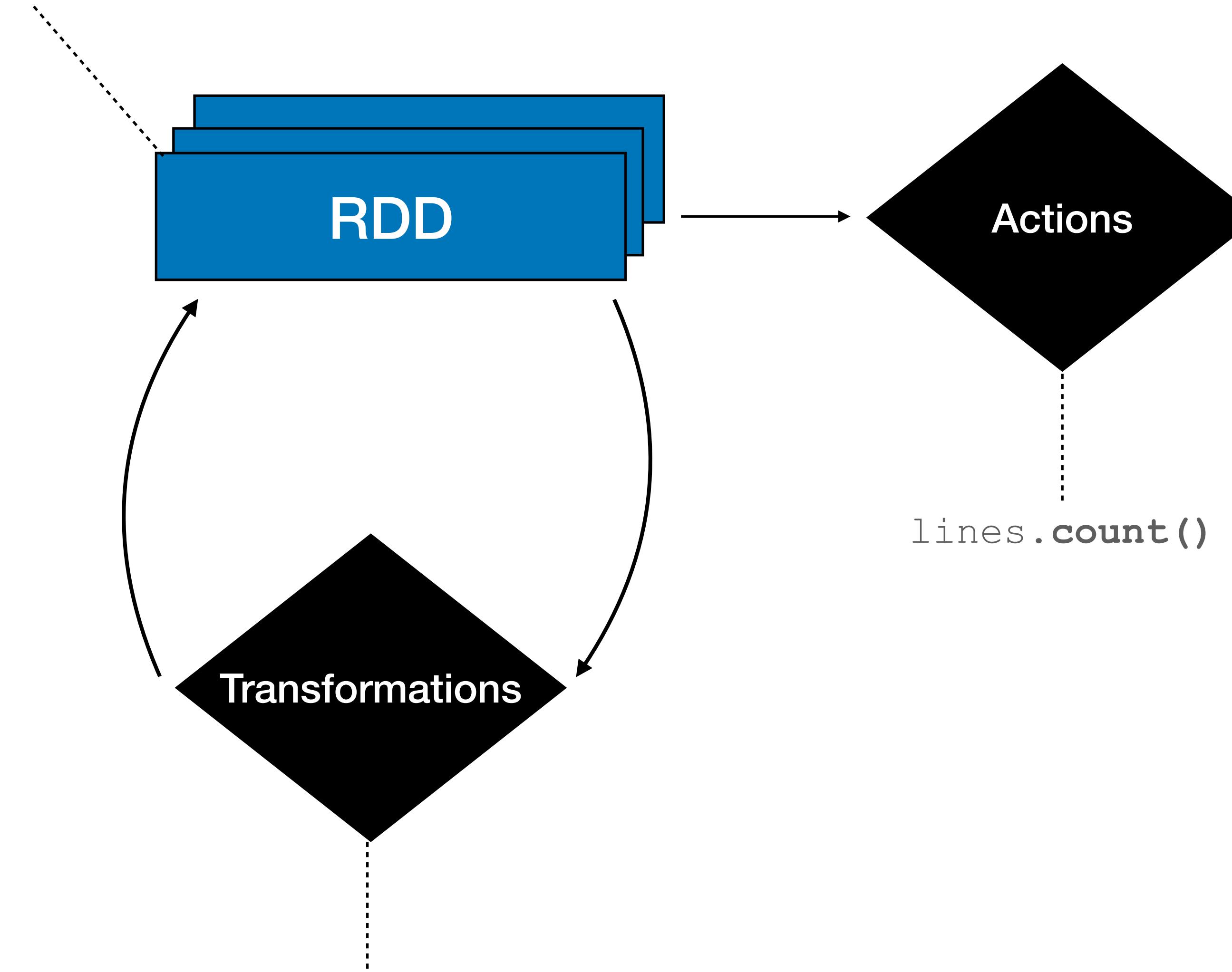


3

```
linesWithITU = lines.filter(line -> line.contains("ITU"))
```

Working with RDDs (I)

```
lines = sc.textFile("hdfs://data.txt")
```

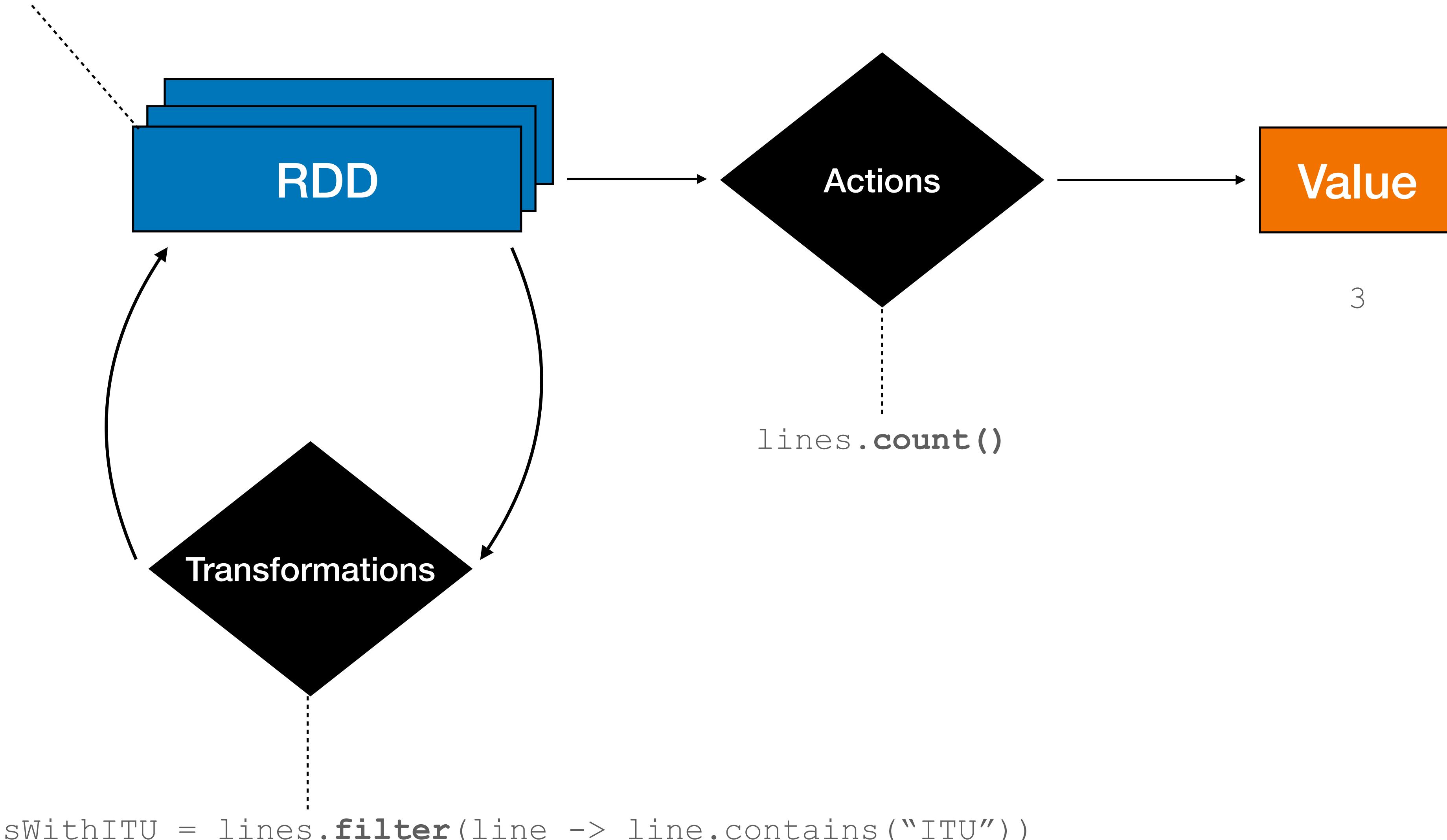


3

```
linesWithITU = lines.filter(line -> line.contains("ITU"))
```

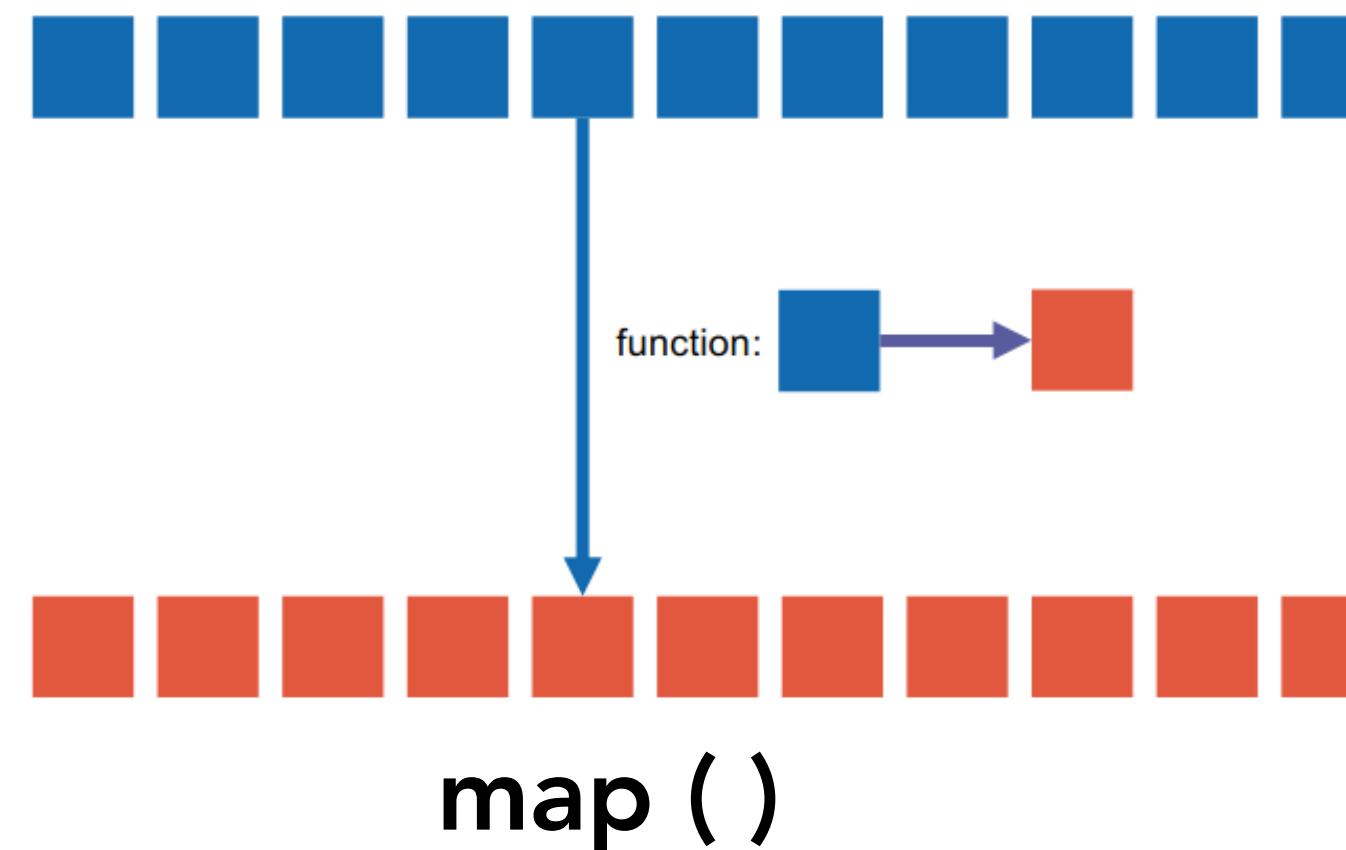
Working with RDDs (I)

```
lines = sc.textFile("hdfs://data.txt")
```

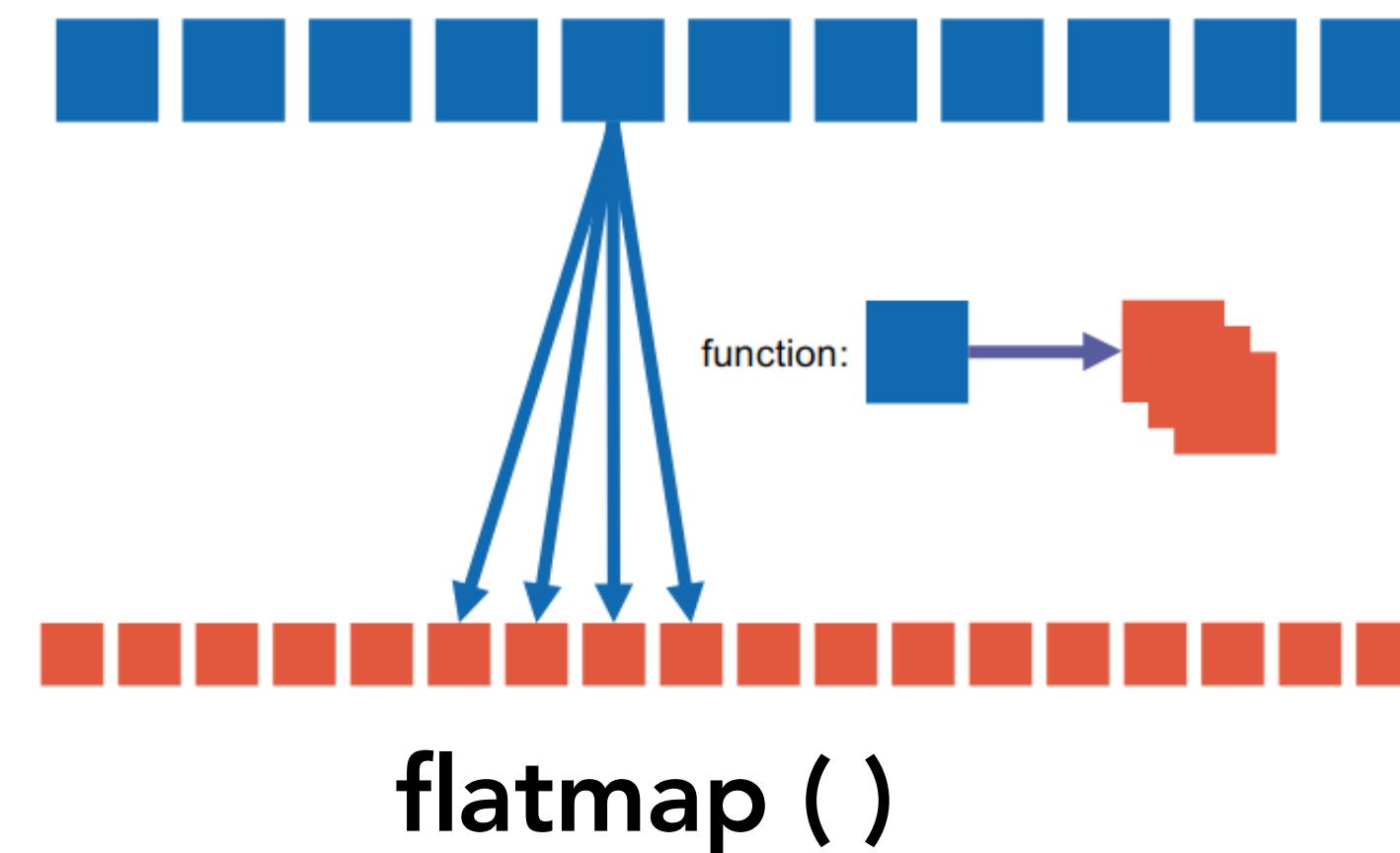
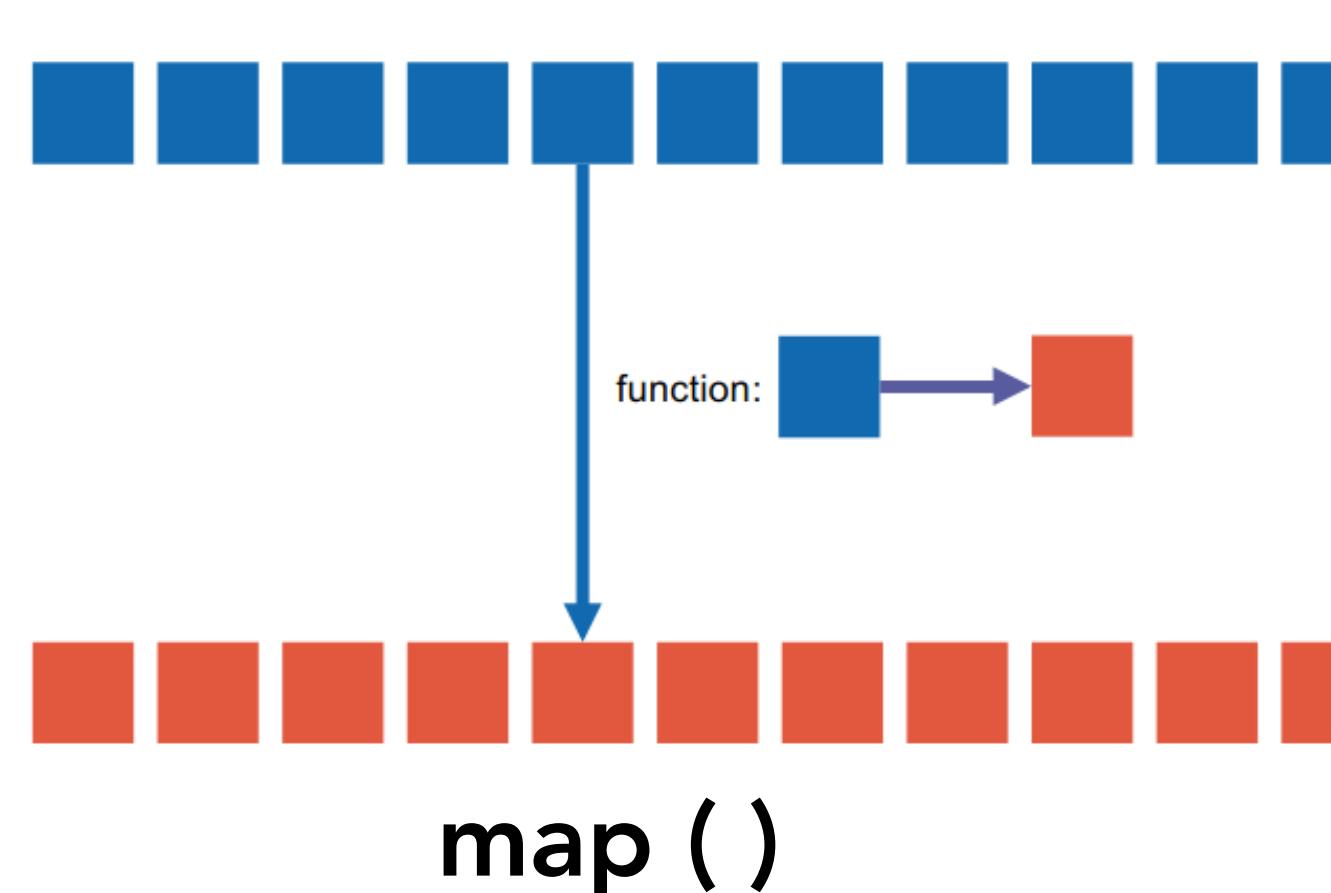


Example transformations

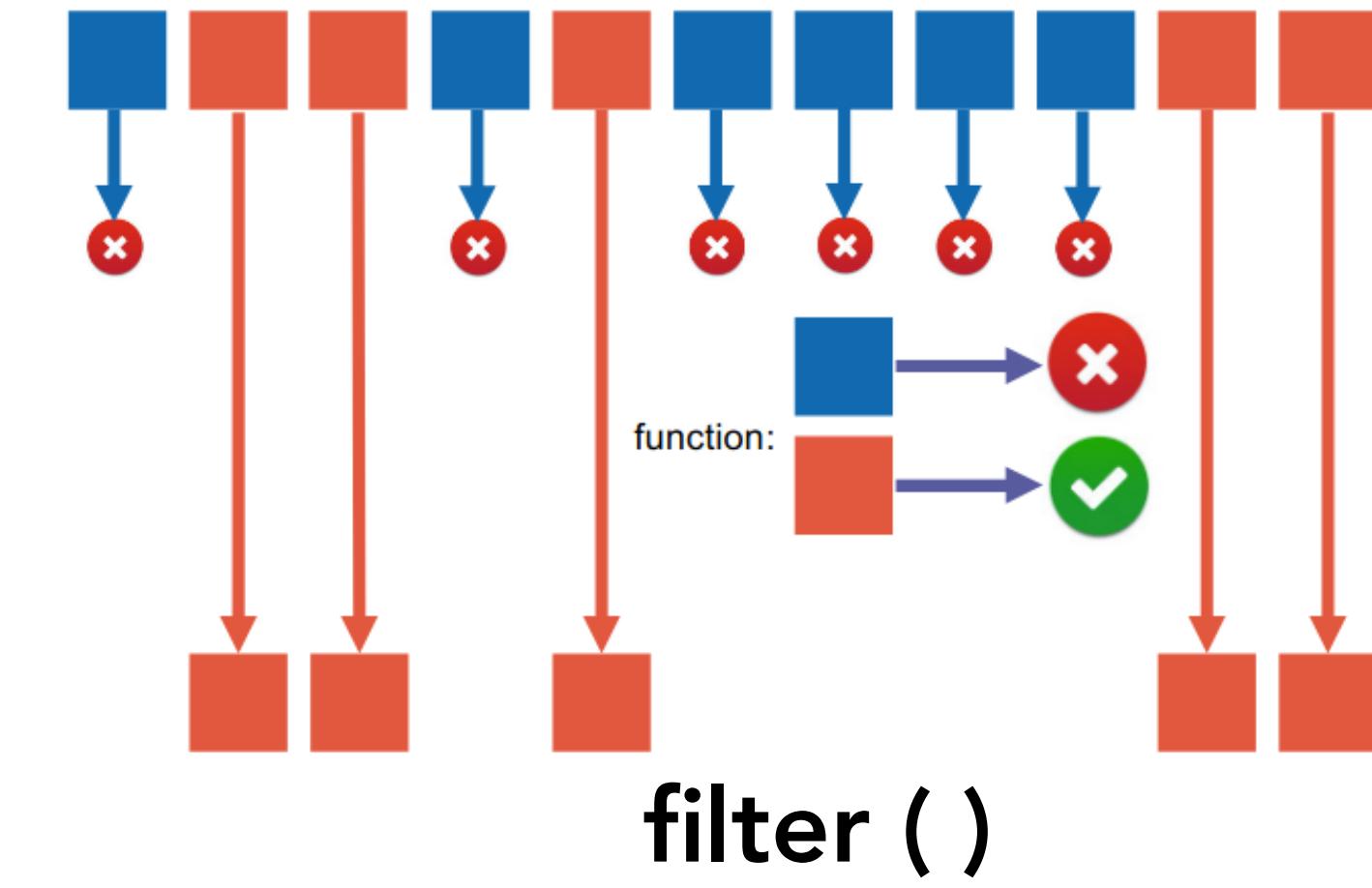
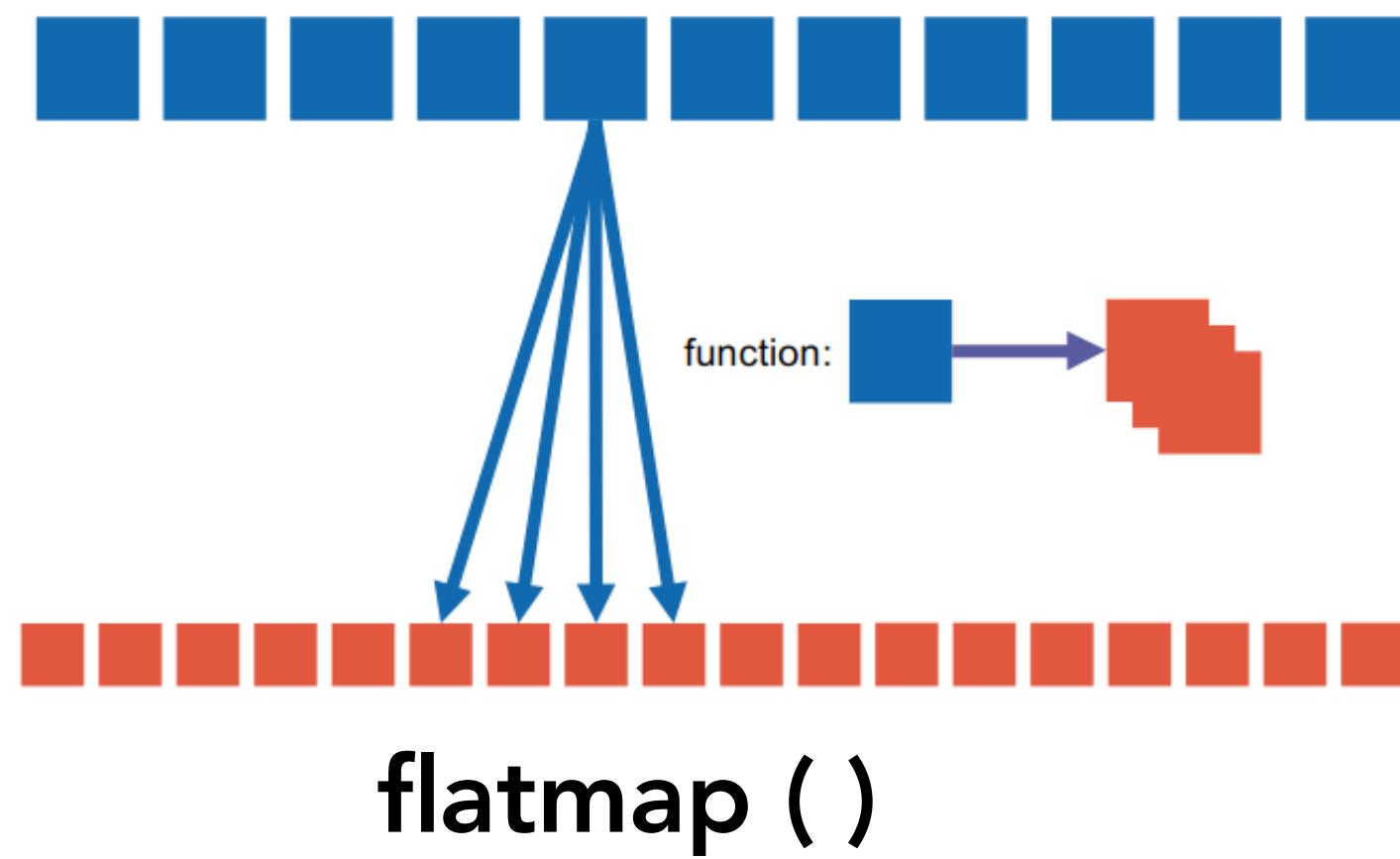
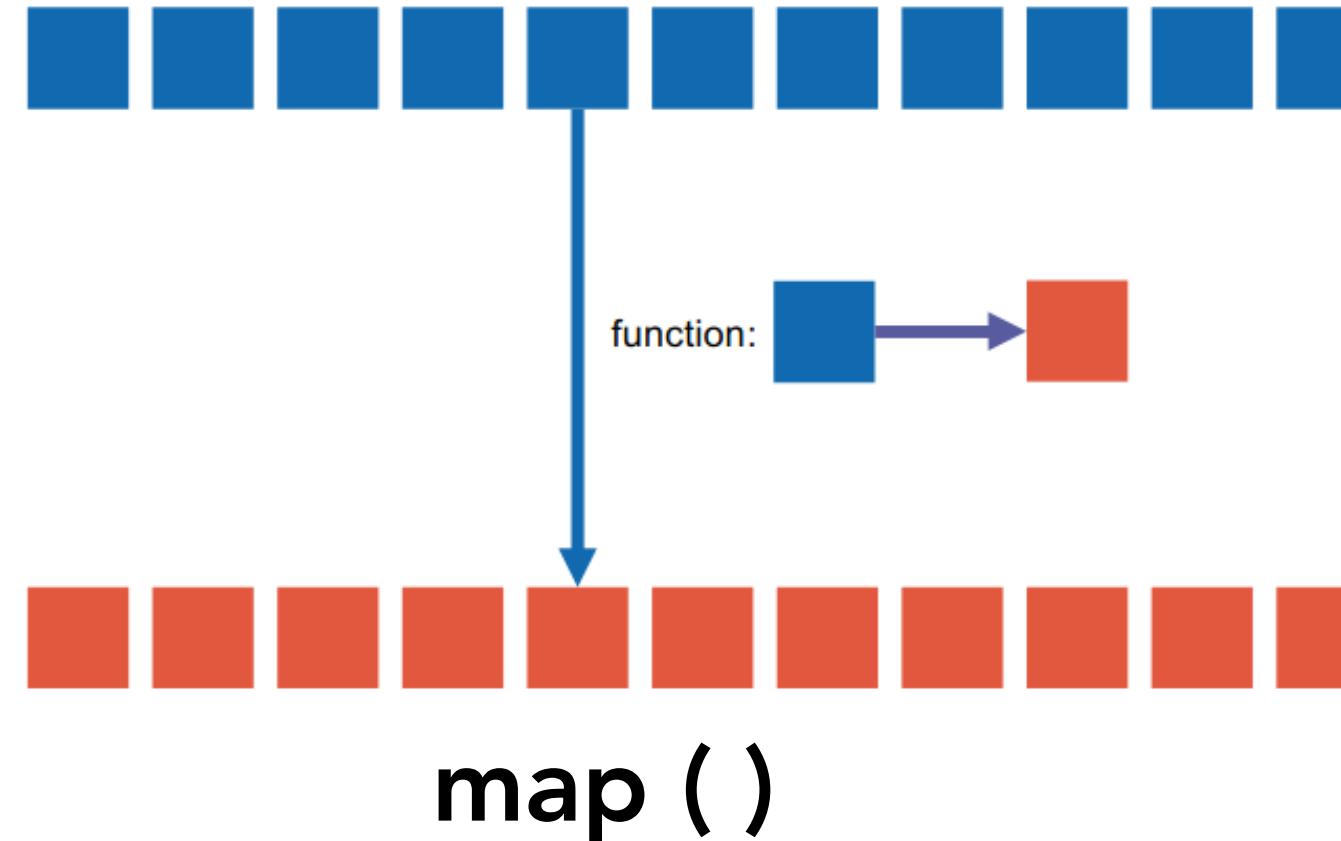
Example transformations



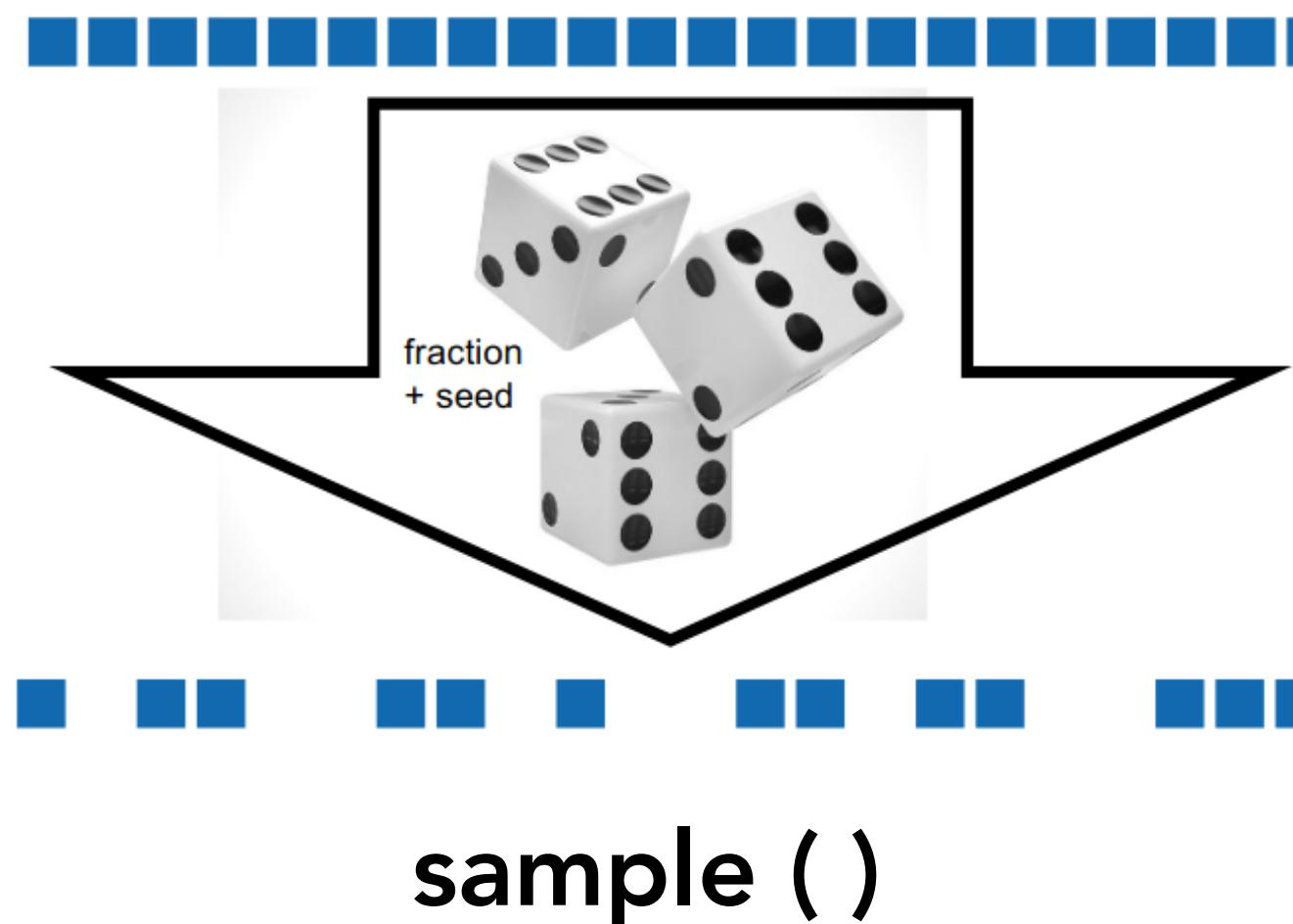
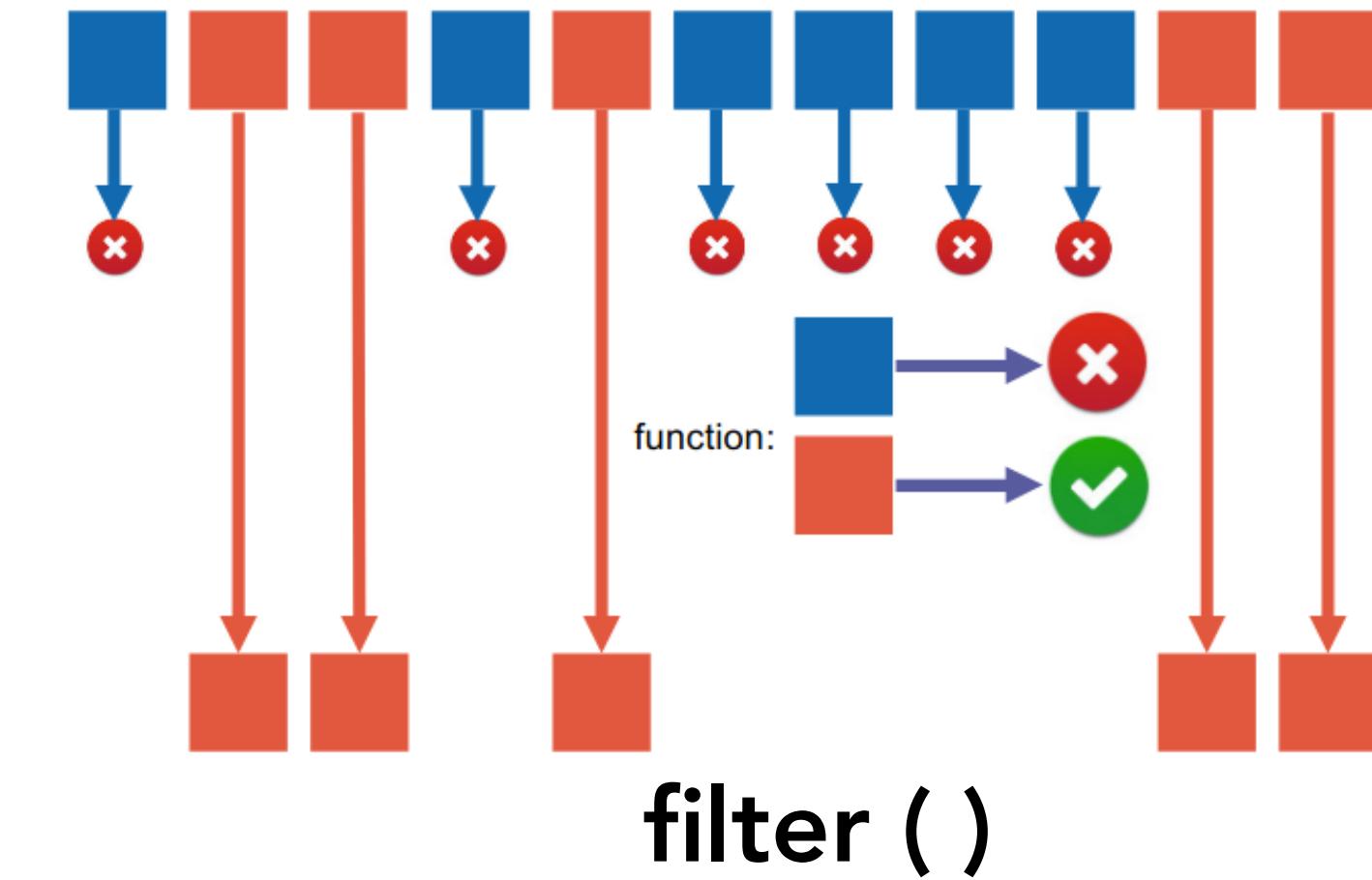
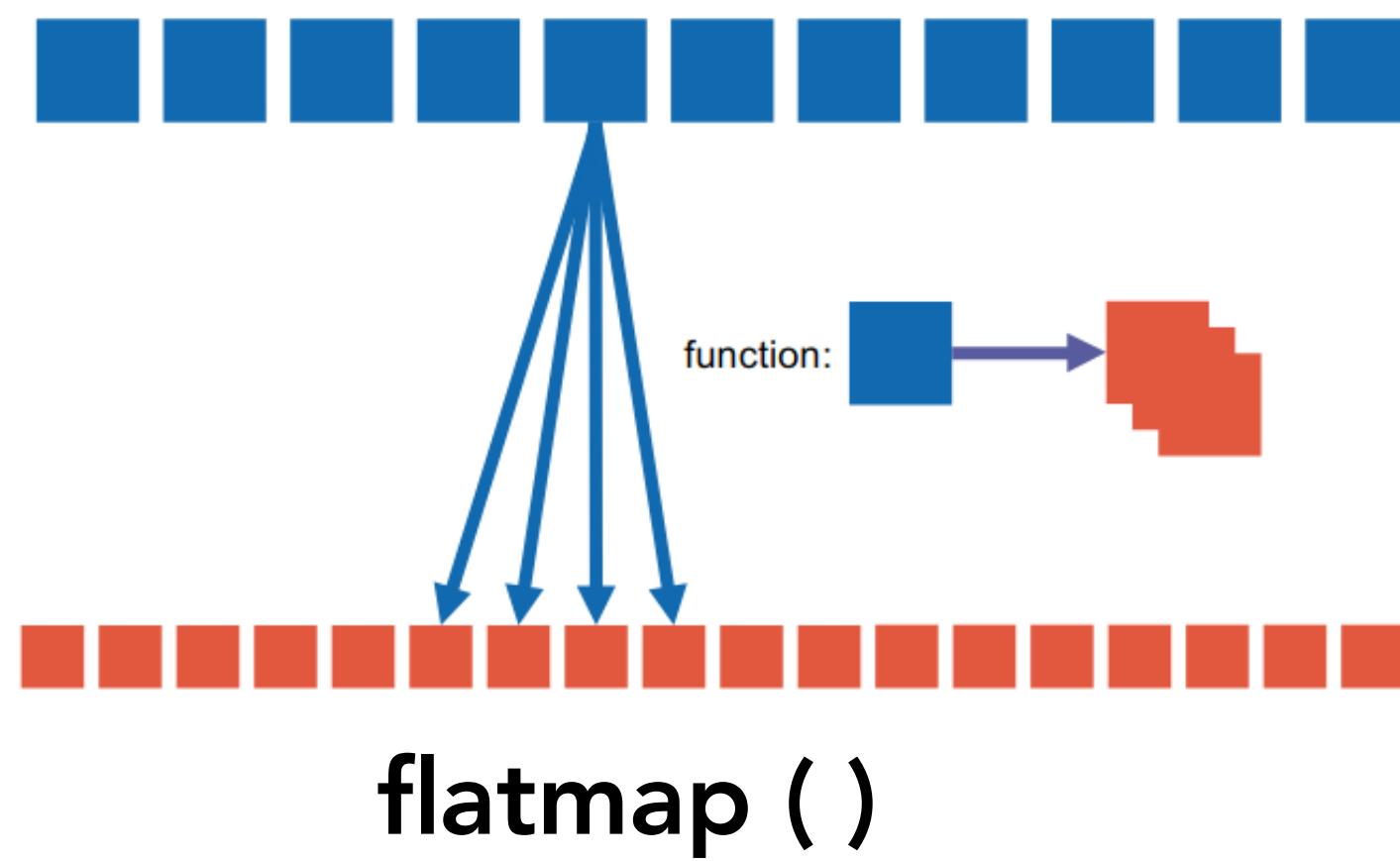
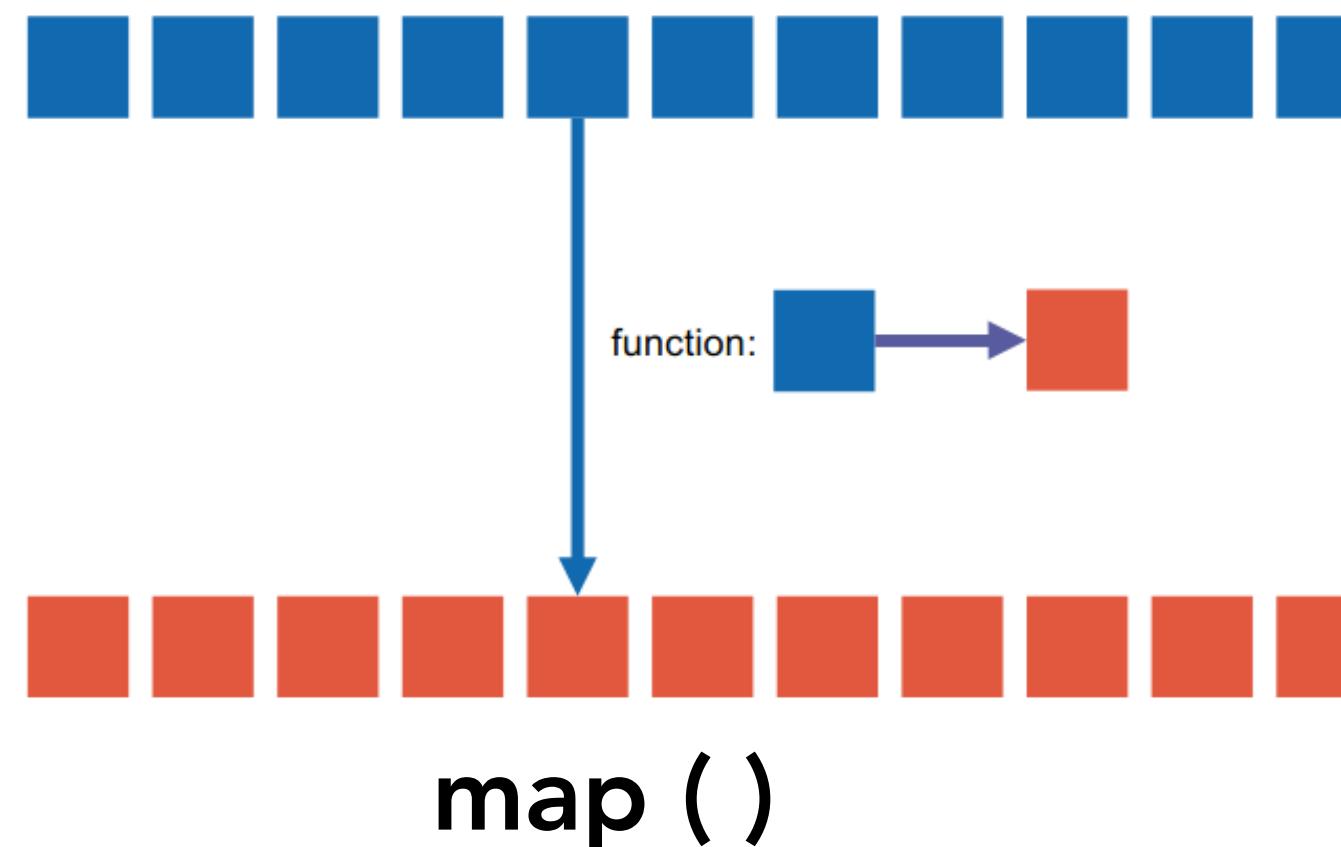
Example transformations



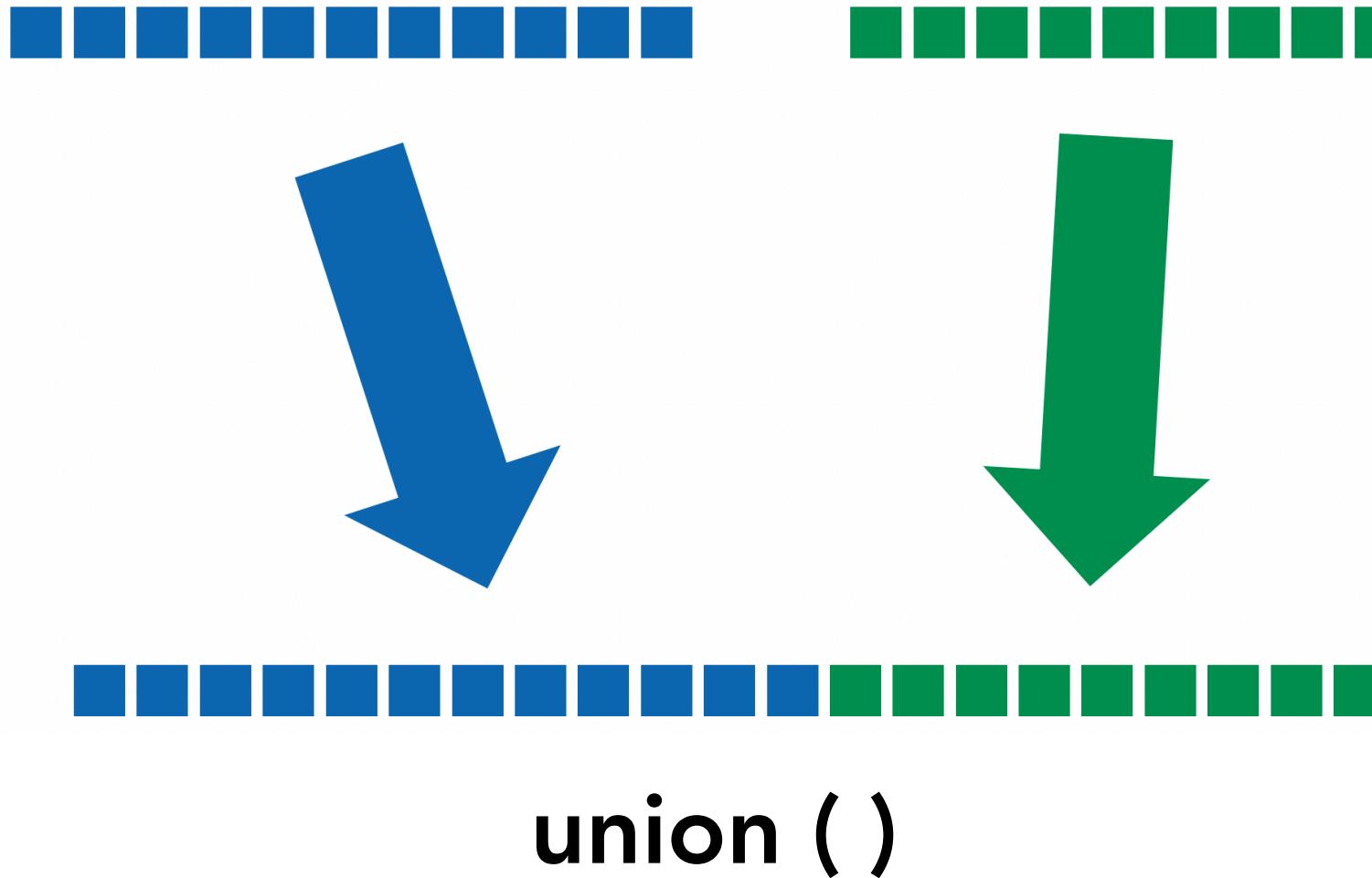
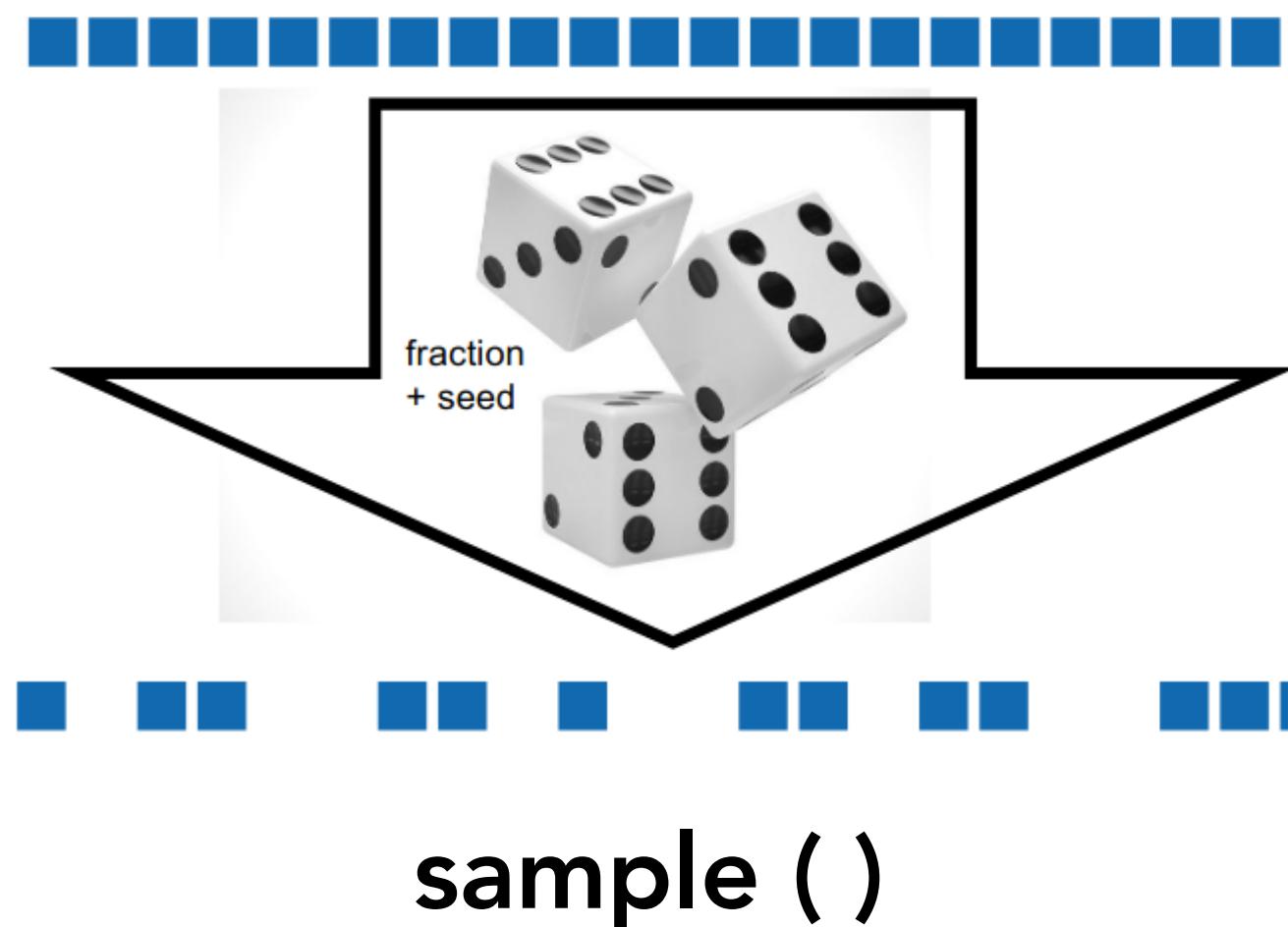
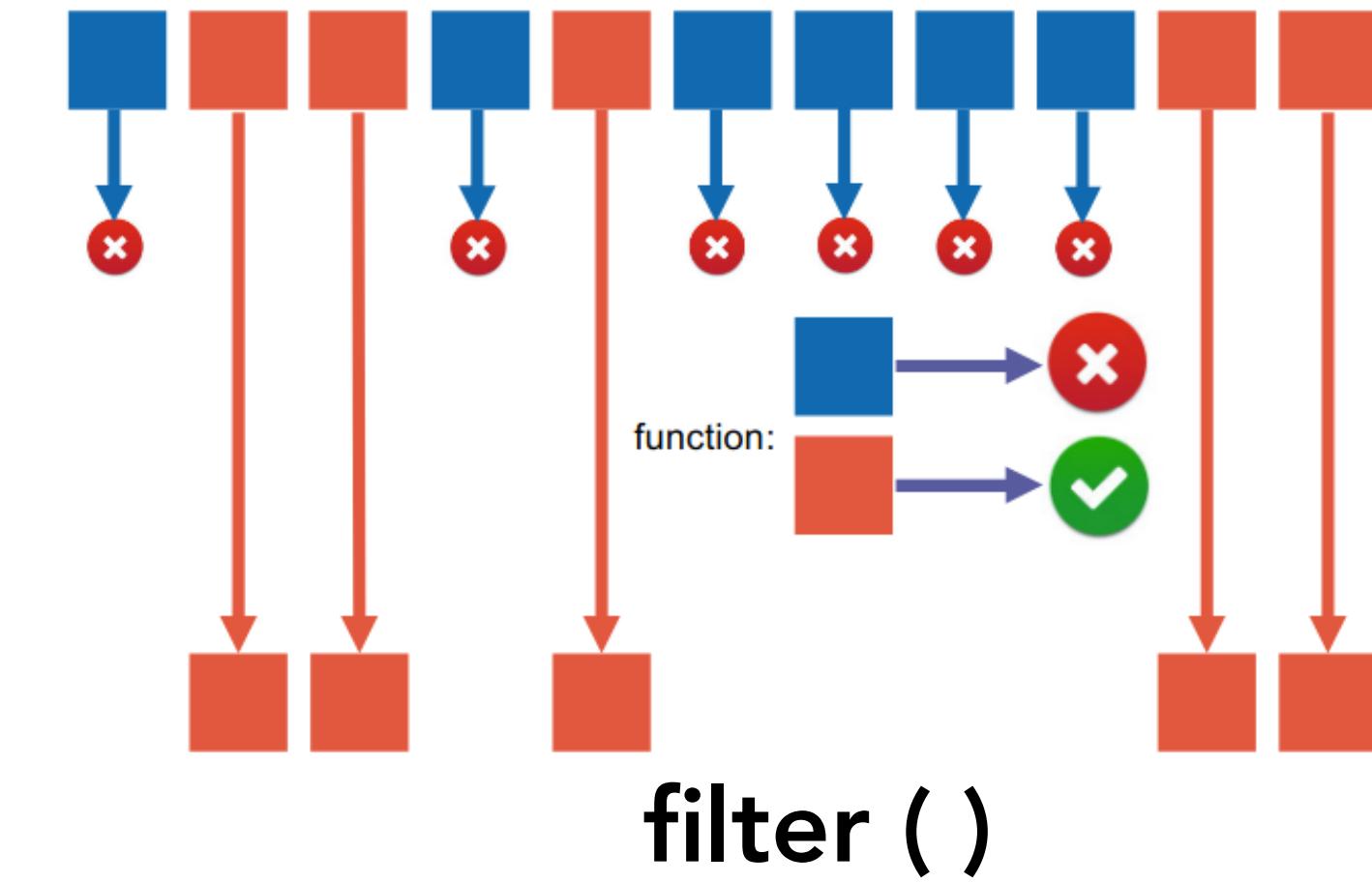
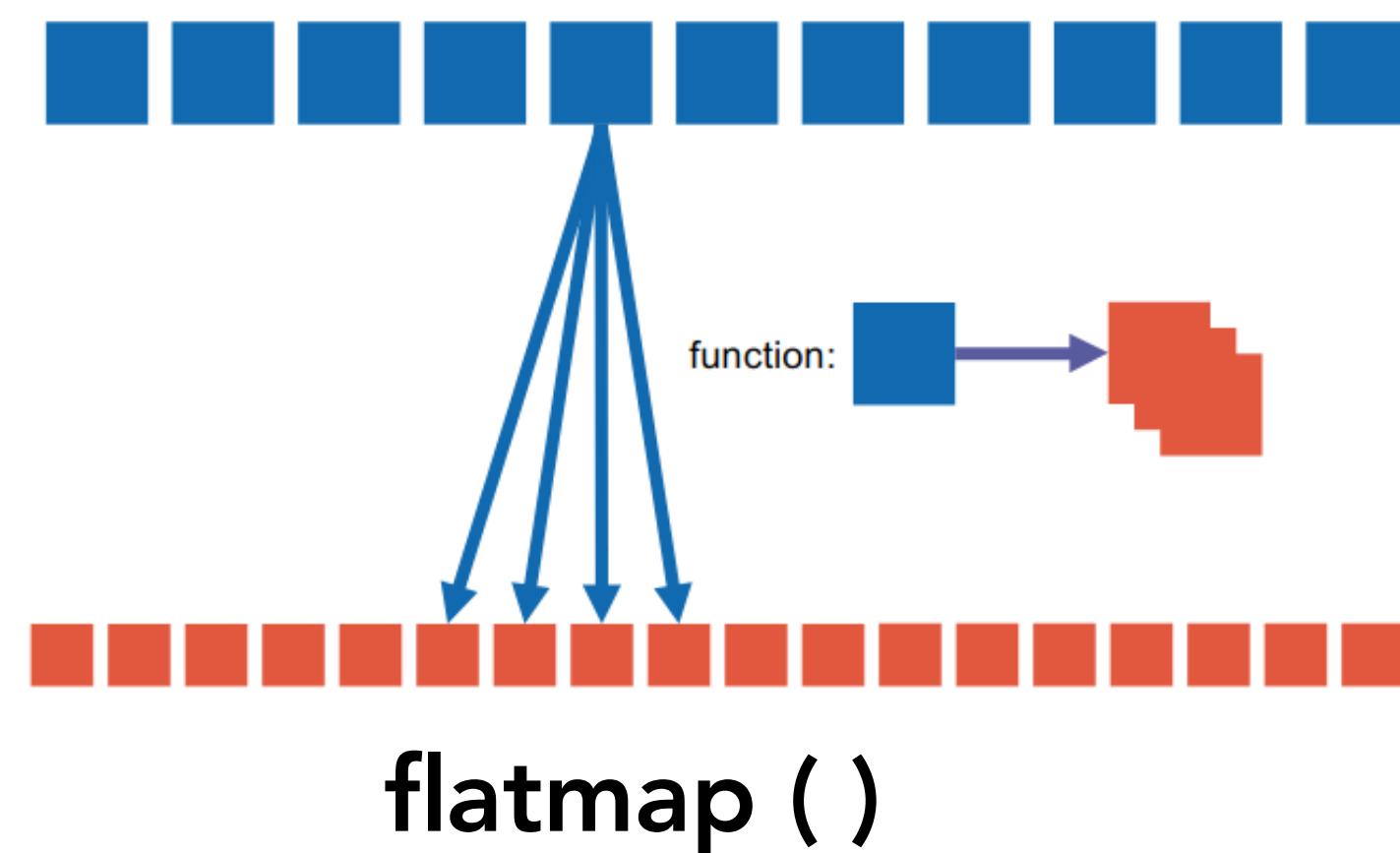
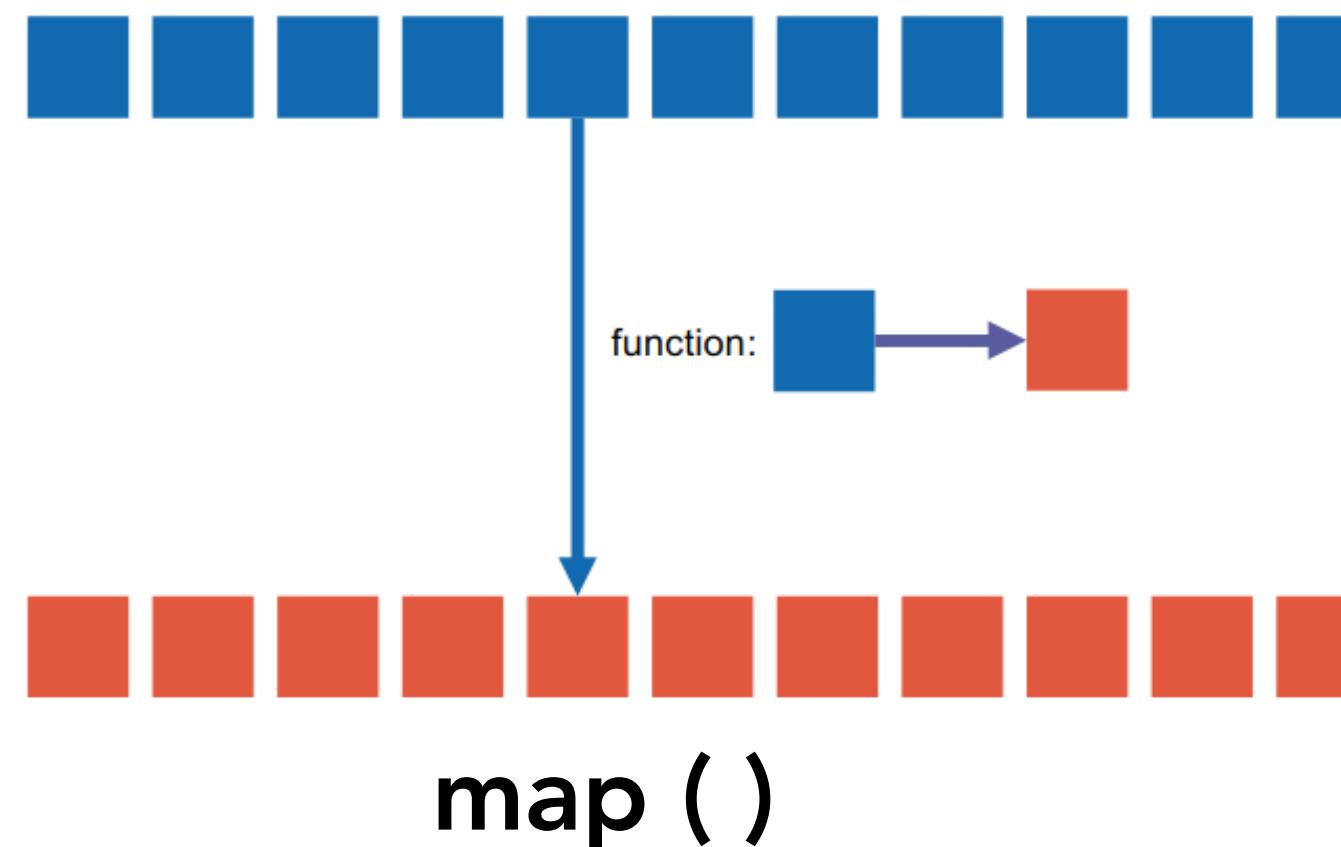
Example transformations



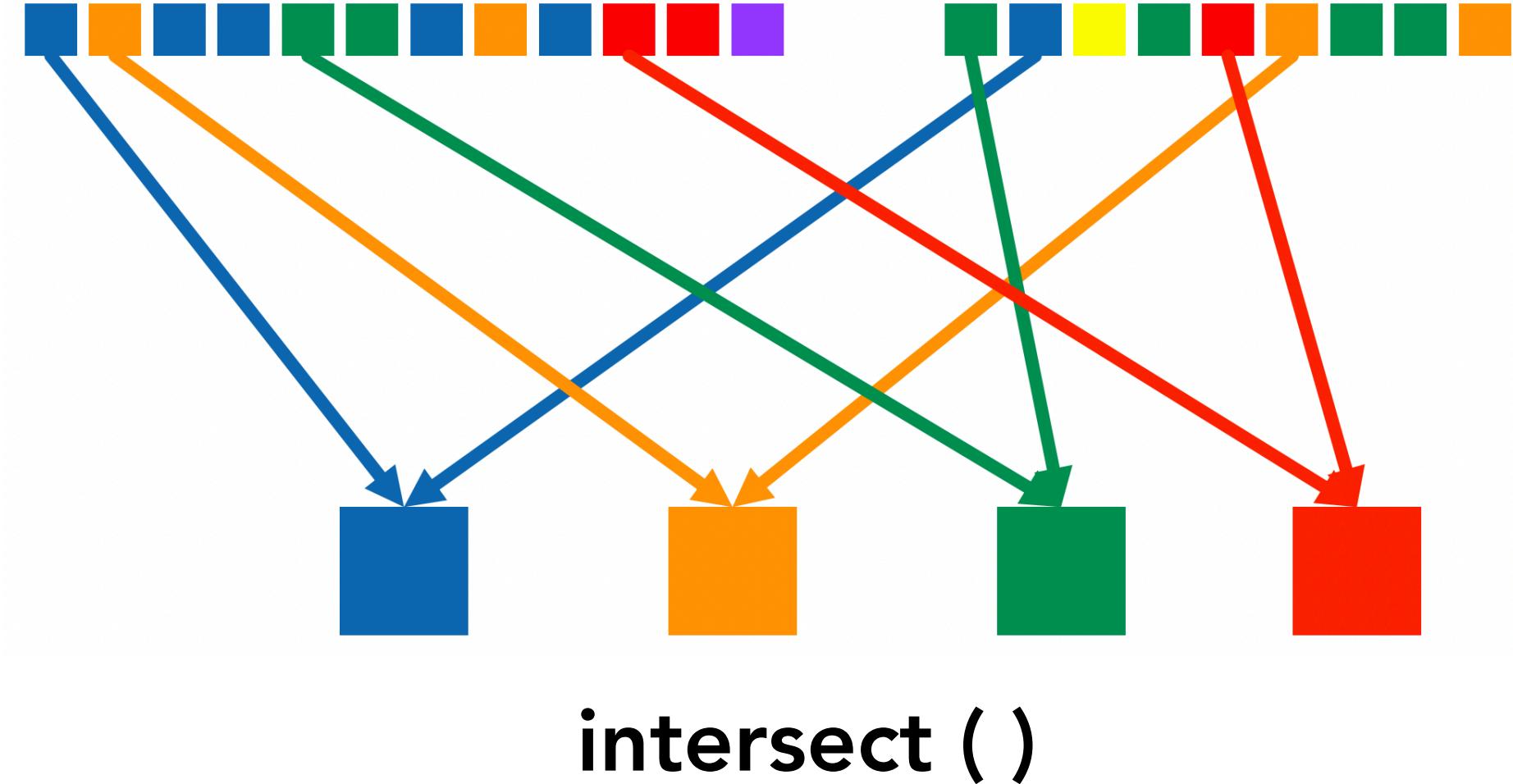
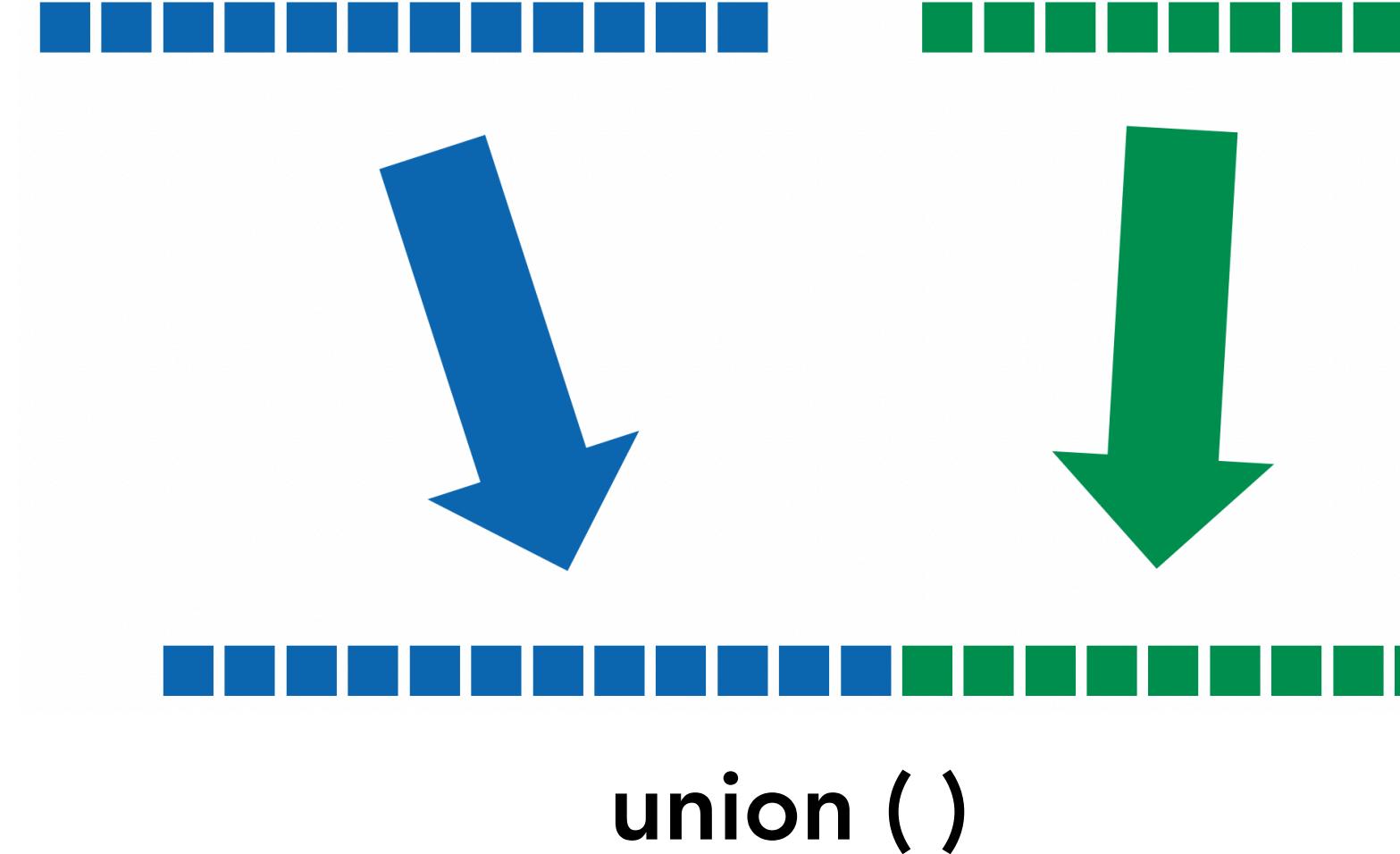
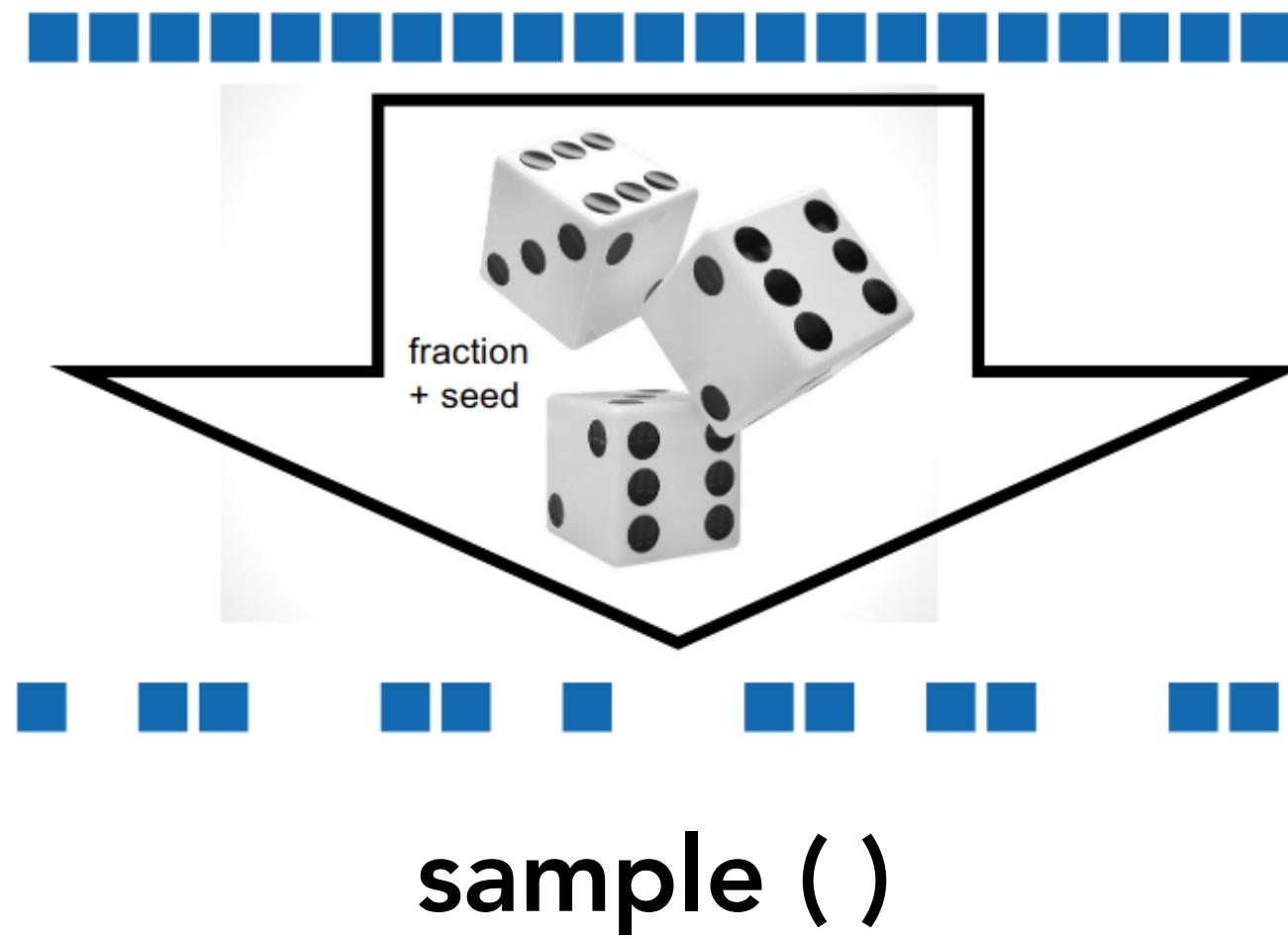
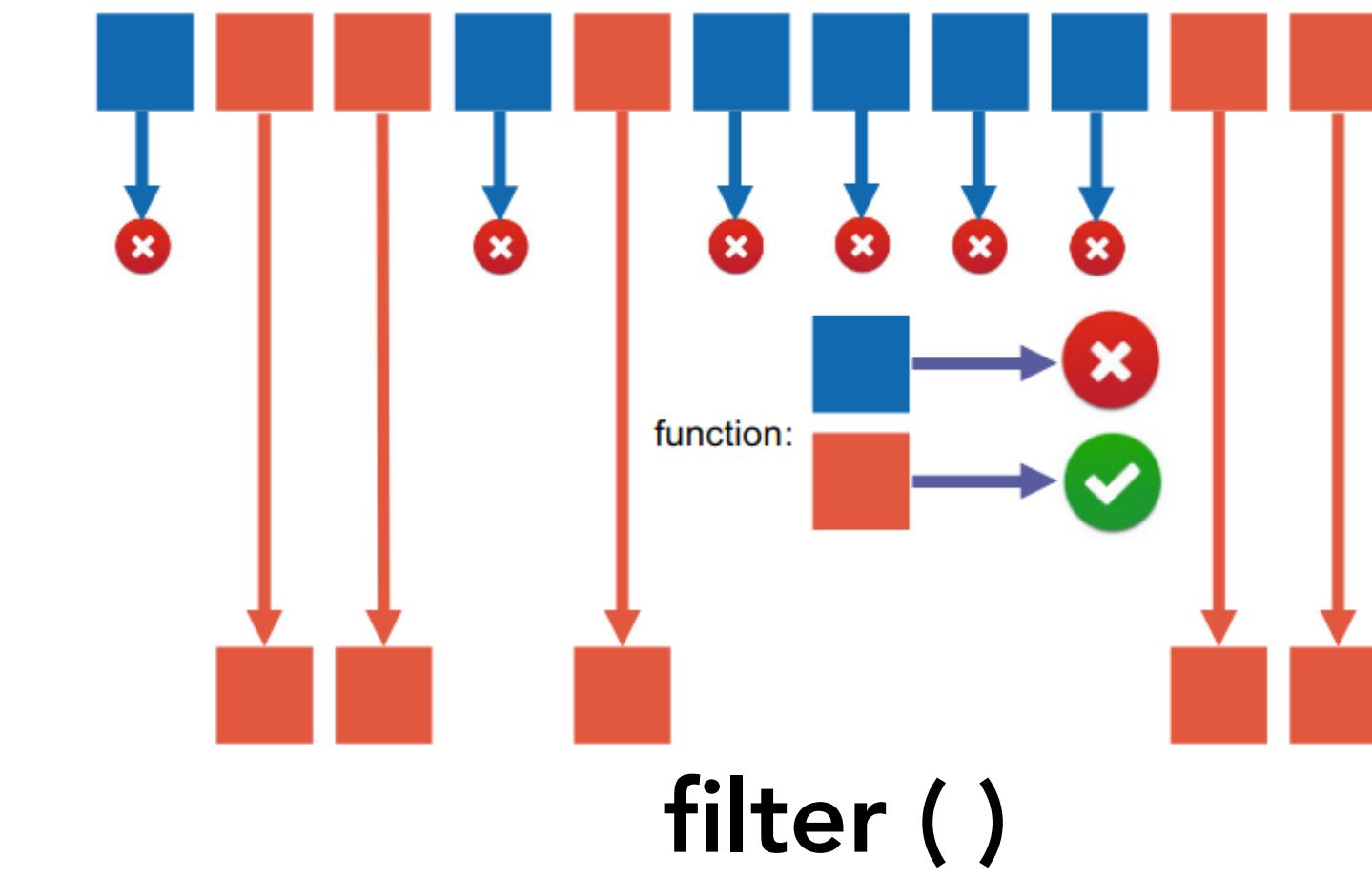
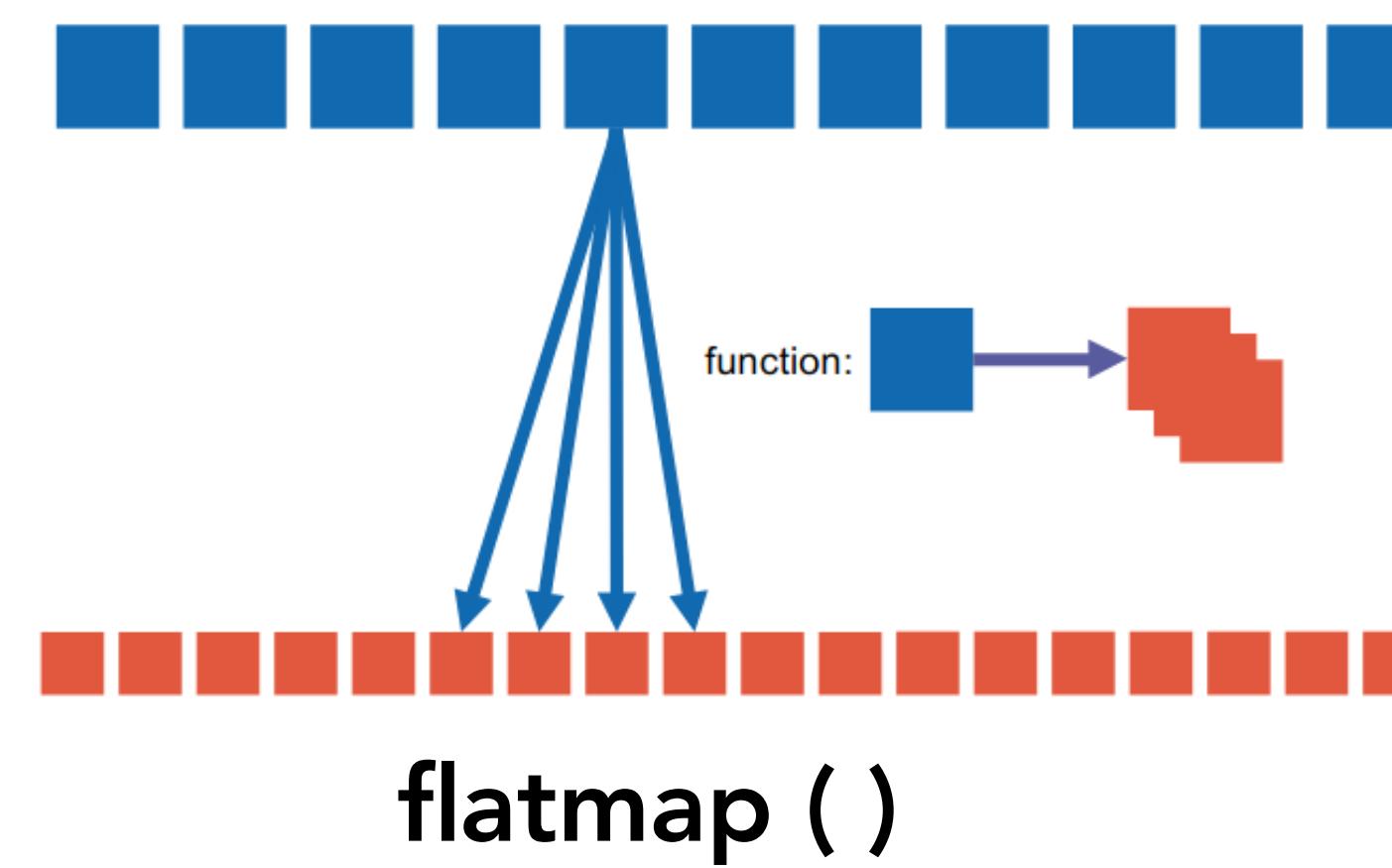
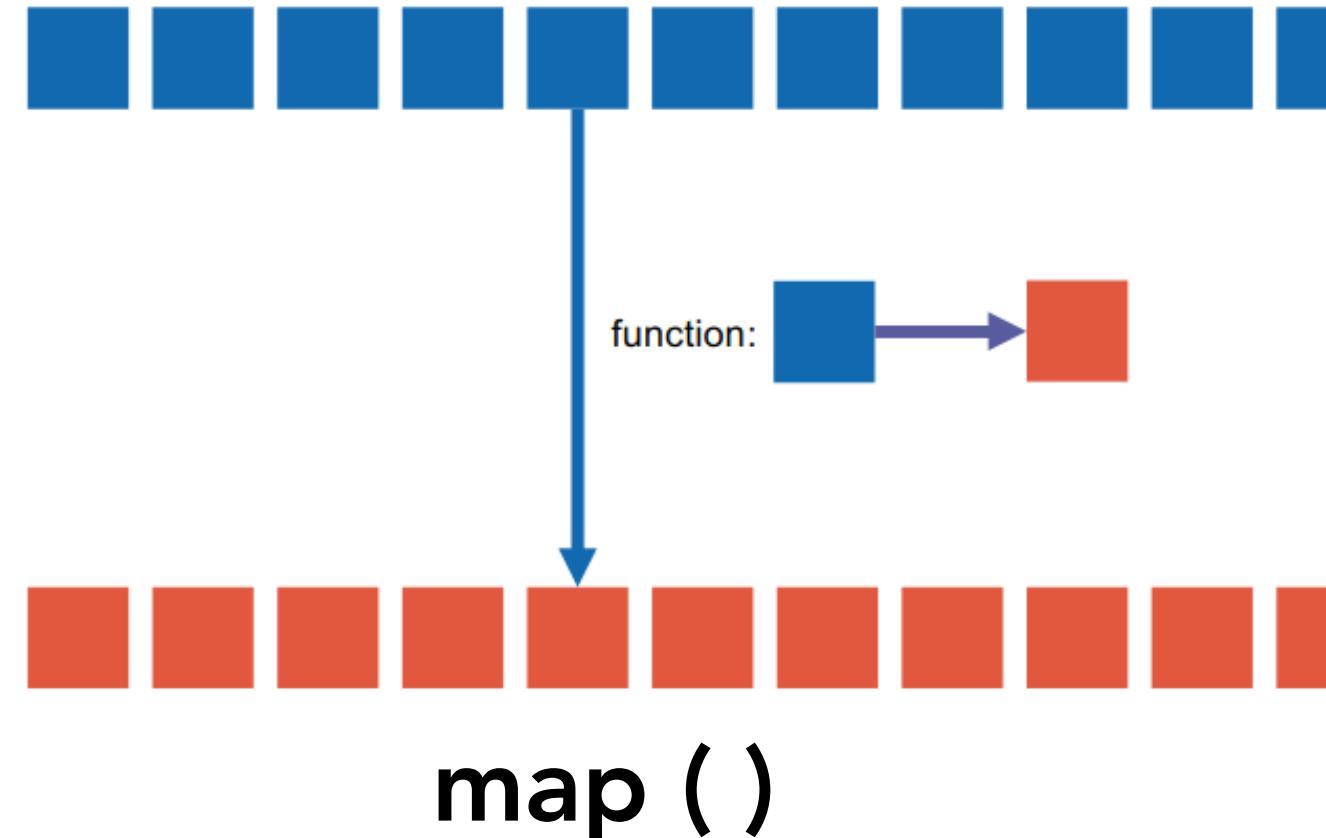
Example transformations



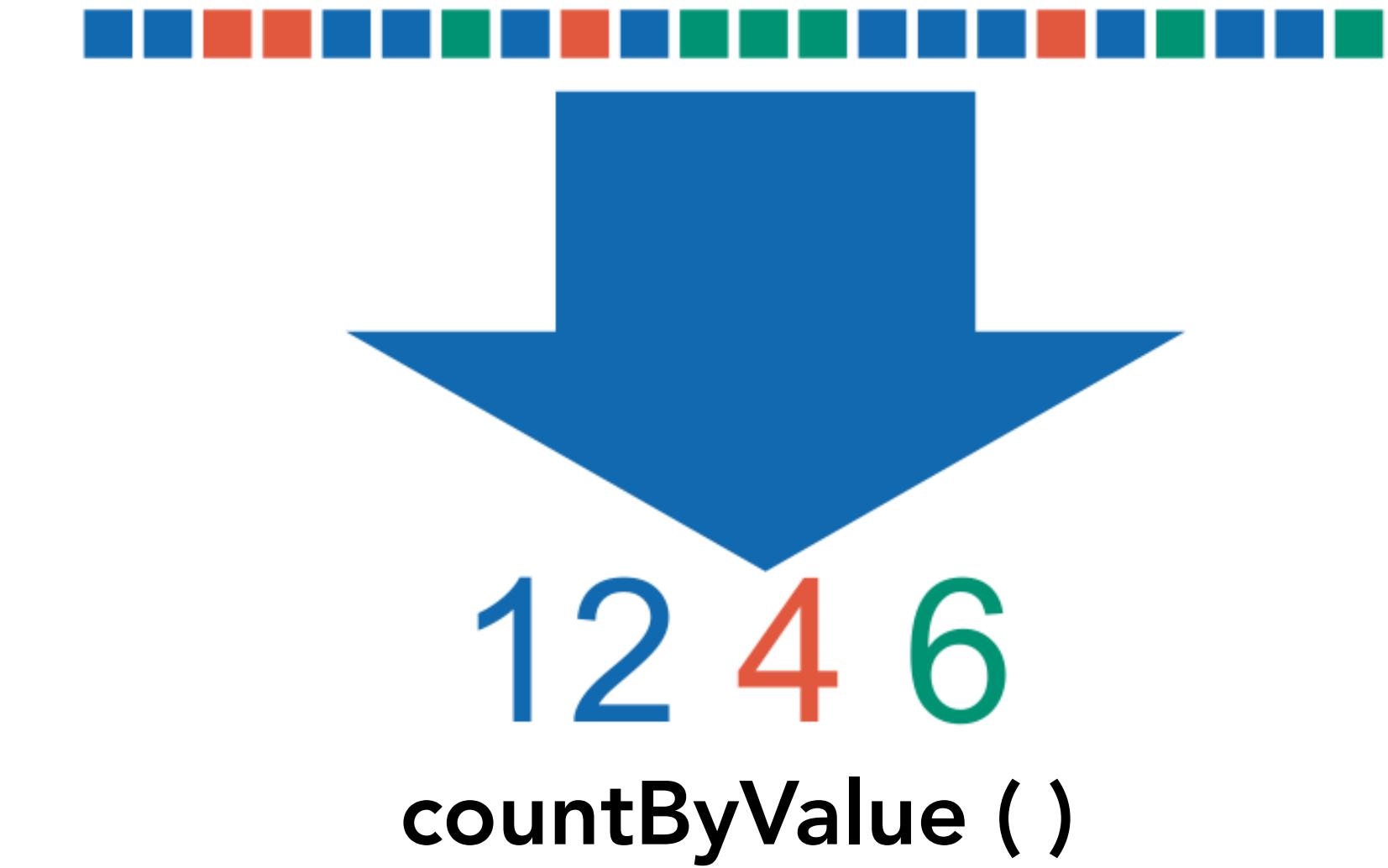
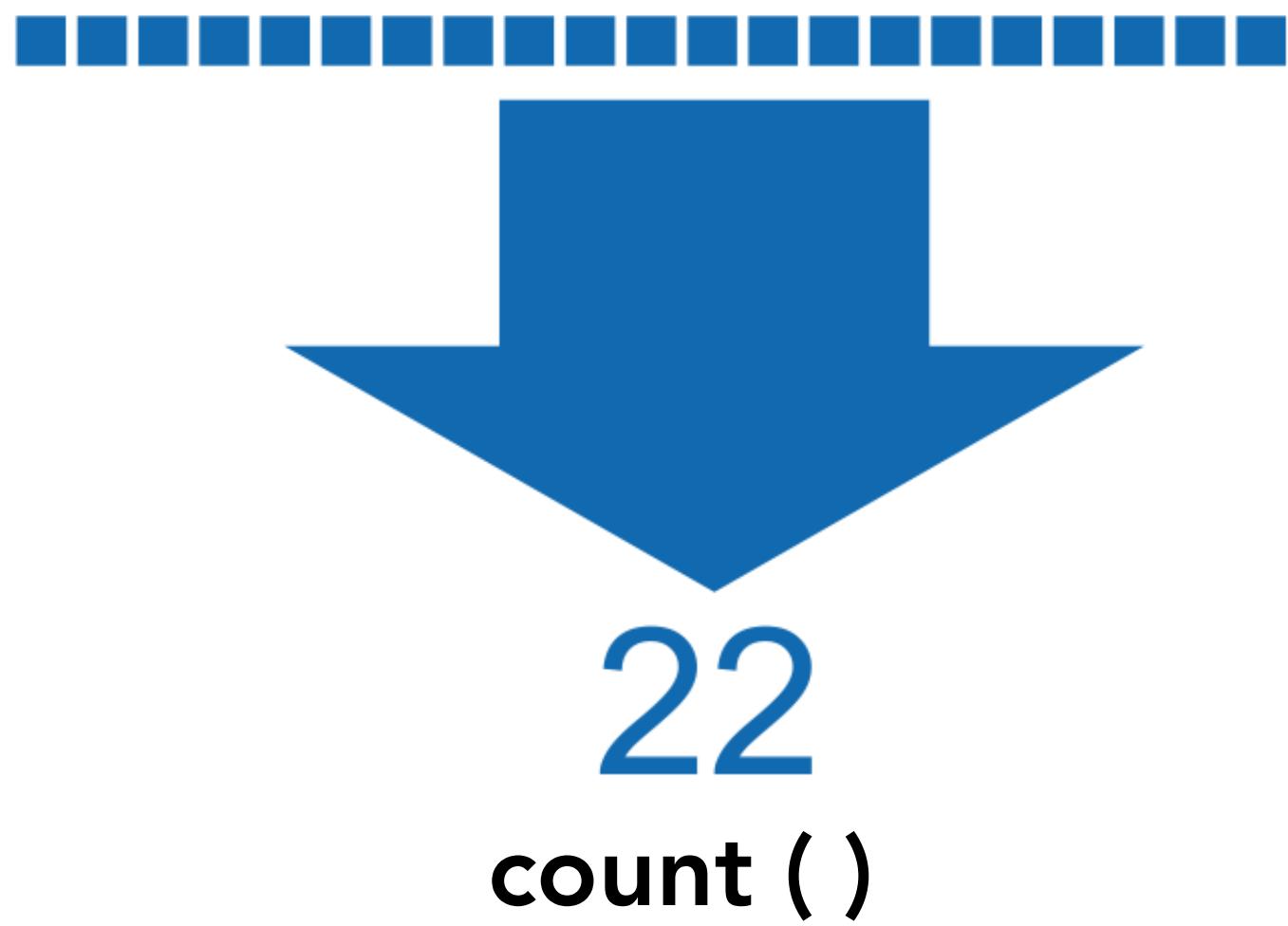
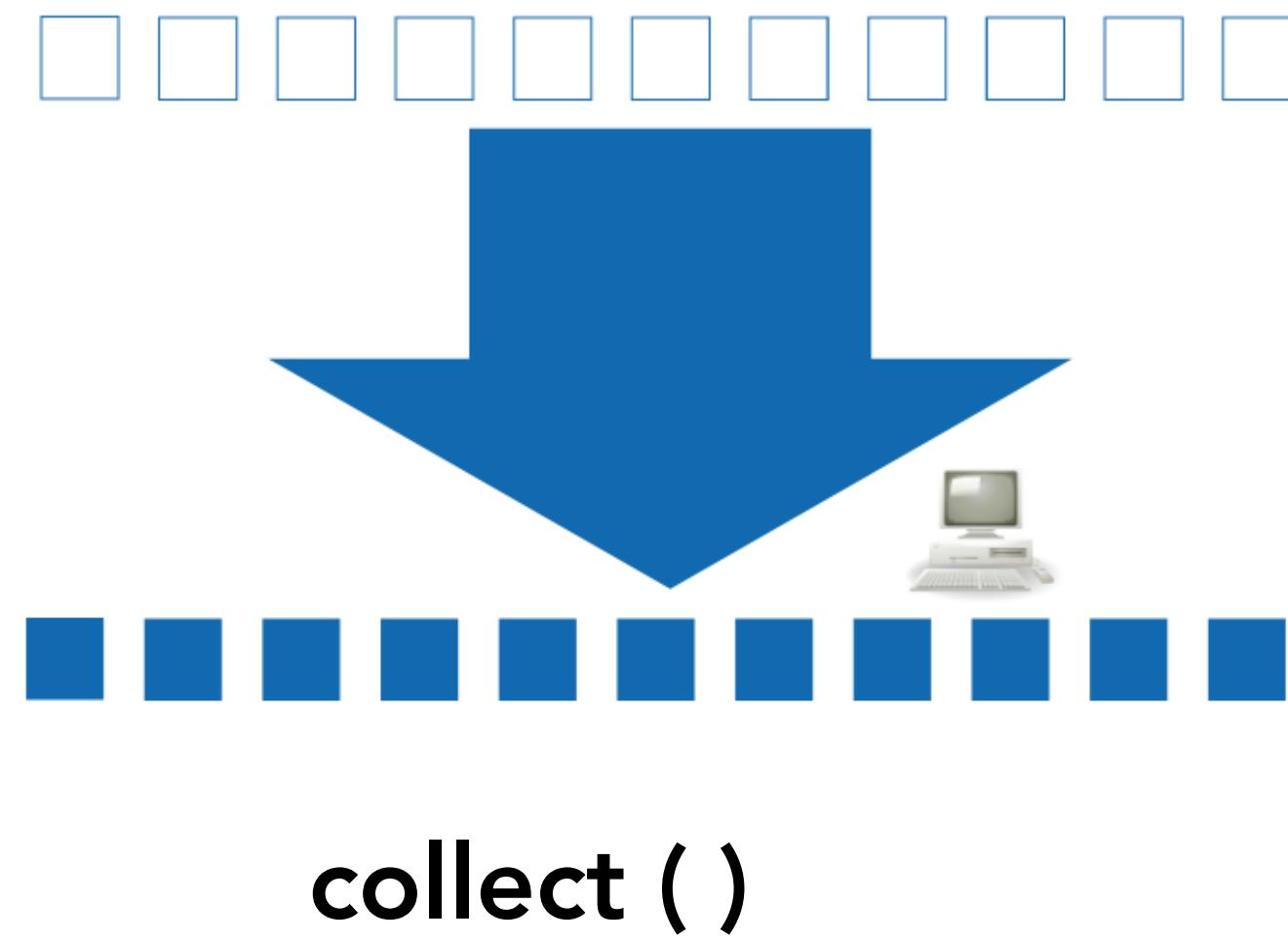
Example transformations



Example transformations



Example of actions



Apache Spark - performance

2013 Record:
Hadoop

2100 machines



72 minutes



2014 Record:
Spark

207 machines



23 minutes



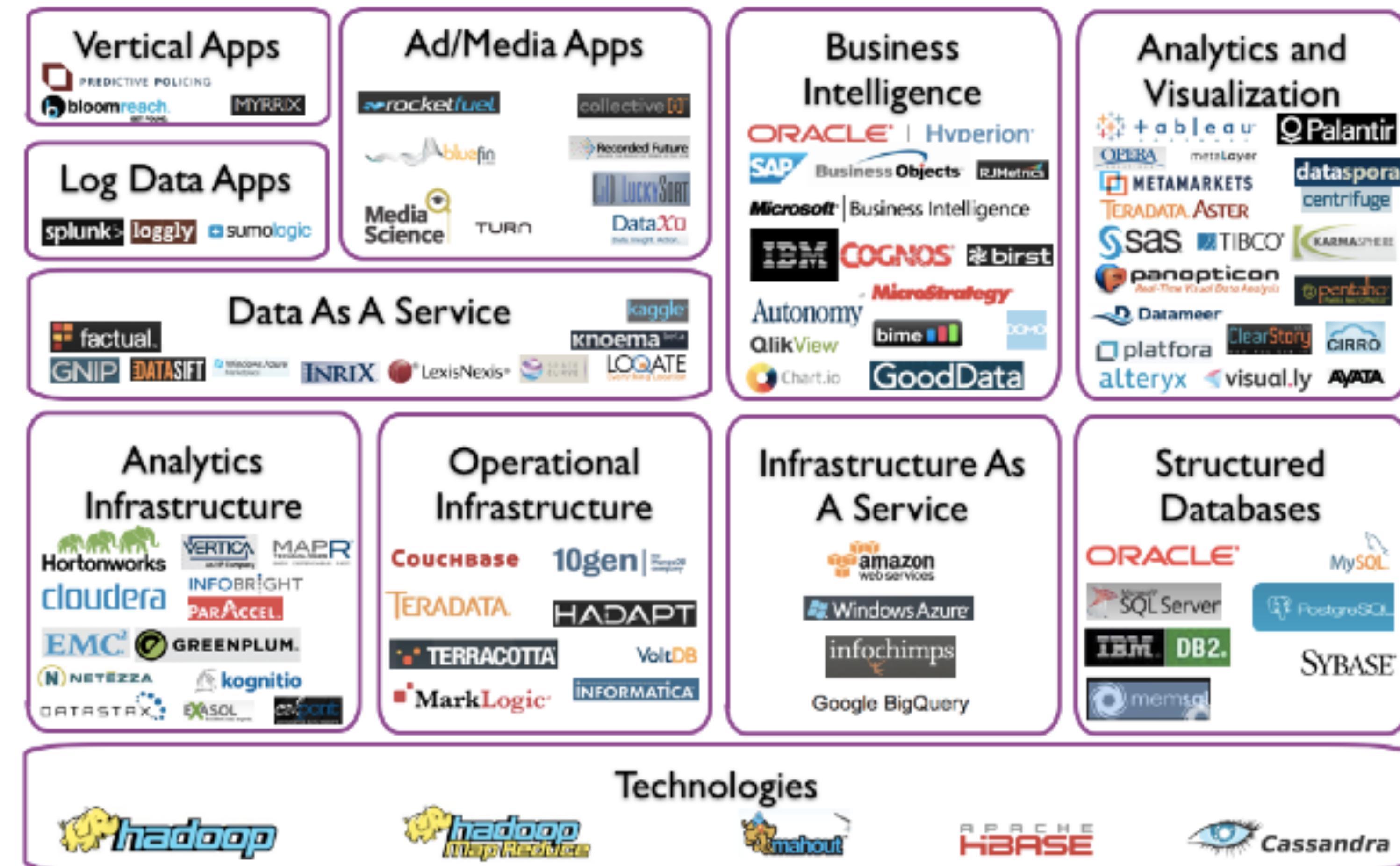
Also sorted 1PB in 4 hours

Readings

- ◆ Chapter 2 from “Mining of Massive Datasets” book
- ◆ GFS paper:
 - ◆ Sanjay Ghemawat et al.: The Google file system. SOSP 2003: 29-43 [[pdf](#)]
- ◆ MapReduce paper:
 - ◆ Jeffrey Dean, Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004: 137-150
- ◆ Spark papers:
 - ◆ Matei Zaharia et al.: Spark: Cluster Computing with Working Sets. HotCloud 2010 [[pdf](#)]
 - ◆ Matei Zaharia et al.: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI 2012: 15-28 [[pdf](#)]

Cross-Platform

Big Data Landscape in 2014



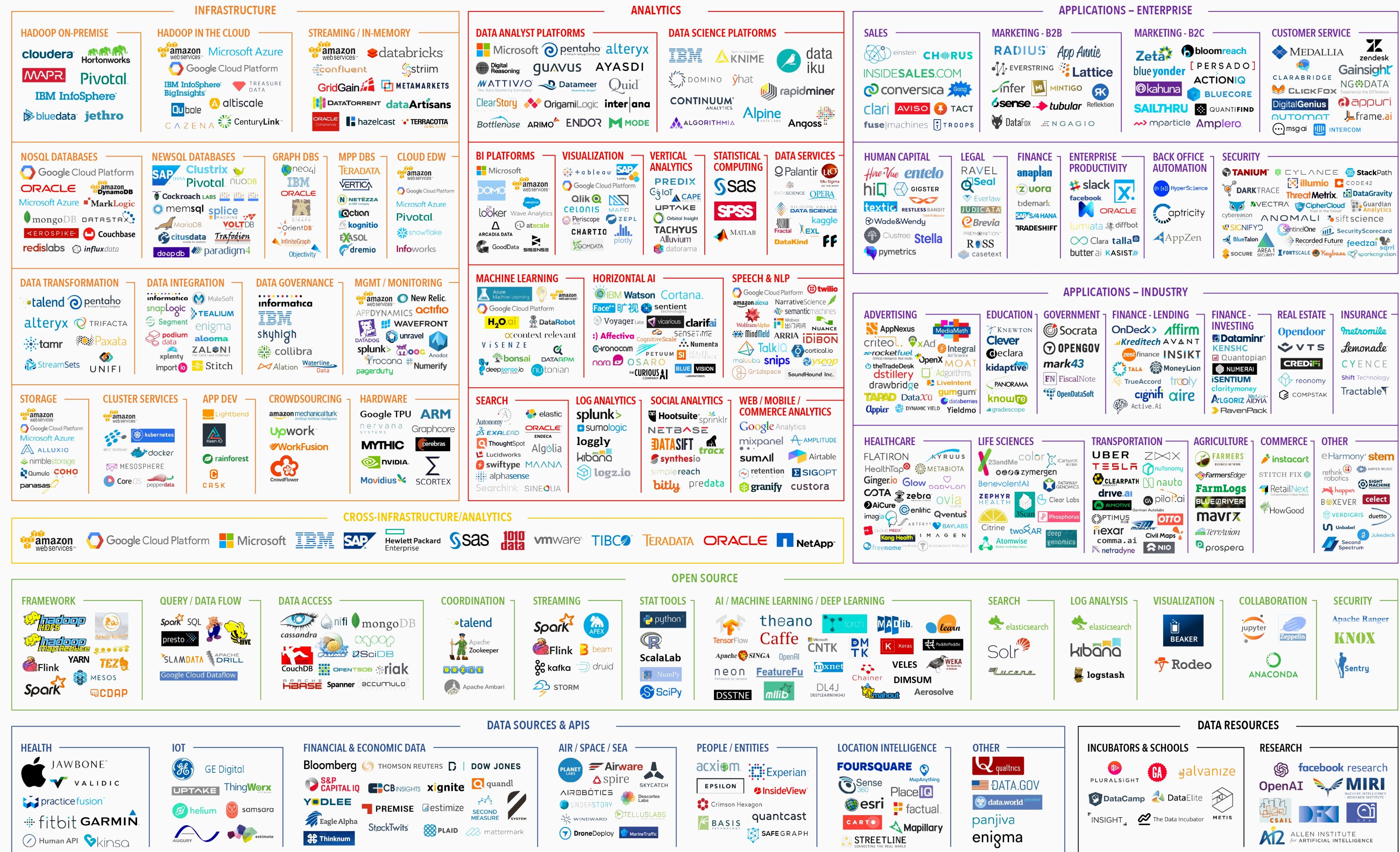
Copyright © 2012 Dave Feinleib

dave@vcdave.com

blogs.forbes.com/davefeinleib

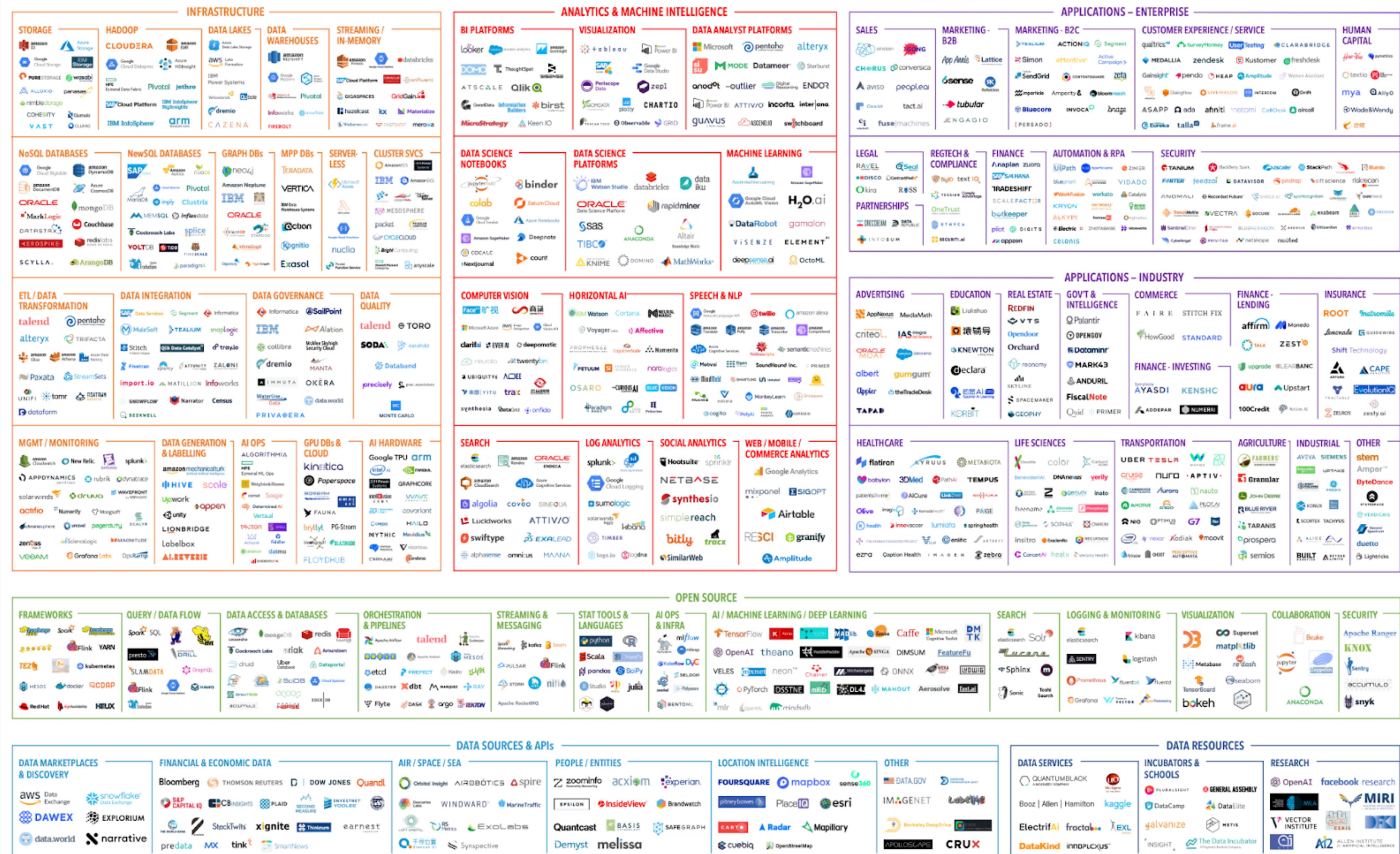
Big Data Landscape in 2017

BIG DATA LANDSCAPE 2017



Big Data Landscape in 2020

DATA & AI LANDSCAPE 2020



Big Data Analytics in 2010

Relational Data

Activity Code	Activity Name
23	Patching
24	Overlay
25	Crack Sealing

Key = 24

Activity Code	Date	Route No.
24	01/12/01	I-95
24	02/08/01	I-66

Date	Activity Code	Route No.
01/12/01	24	I-95
01/15/01	23	I-495
02/08/01	24	I-66

SPJ Queries

```
SELECT name  
FROM employee e  
INNER JOIN Person p  
ON p.firstname = e.name
```

Big Data Analytics in 2020

Relational Data

Activity Code	Activity Name
23	Patching
24	Overlay
25	Crack Sealing

Key = 24

Activity Code	Date	Route No.
---------------	------	-----------

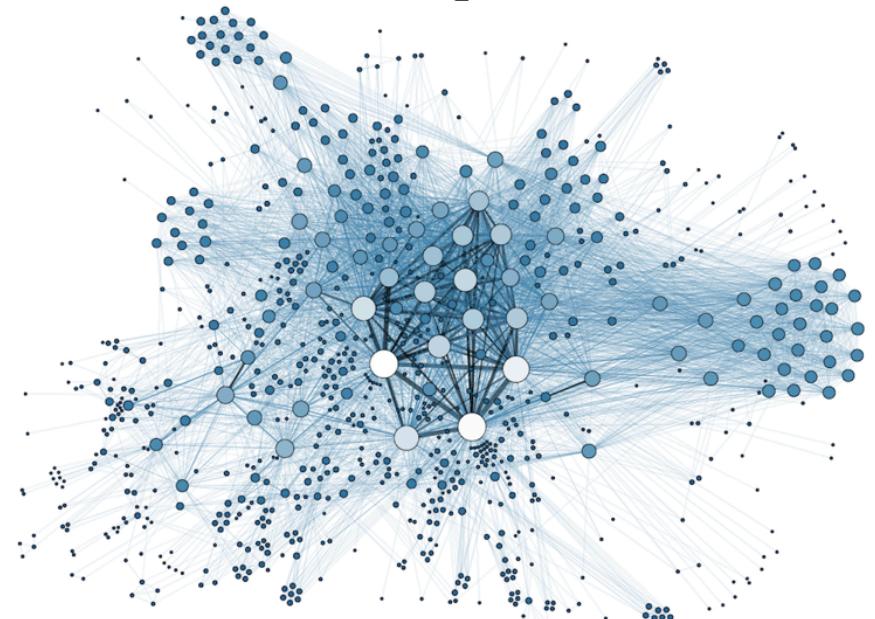
Key = 24

Date	Activity Code	Route No.
01/12/01	24	I-95
01/15/01	23	I-495
02/08/01	24	I-66

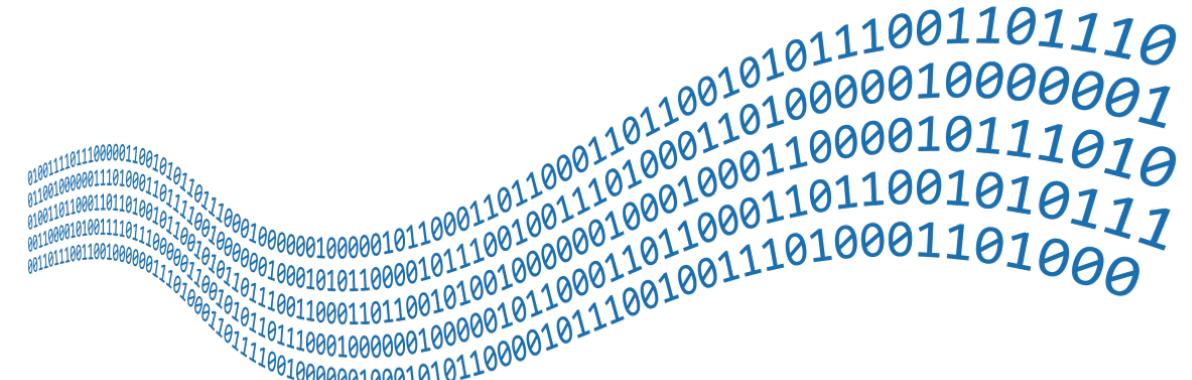
Text Data



Graph Data

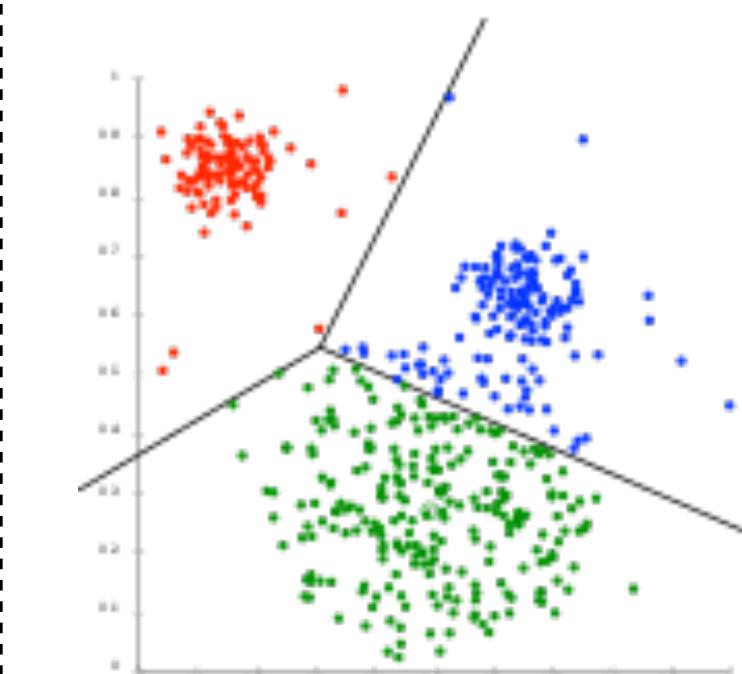


Streaming Data

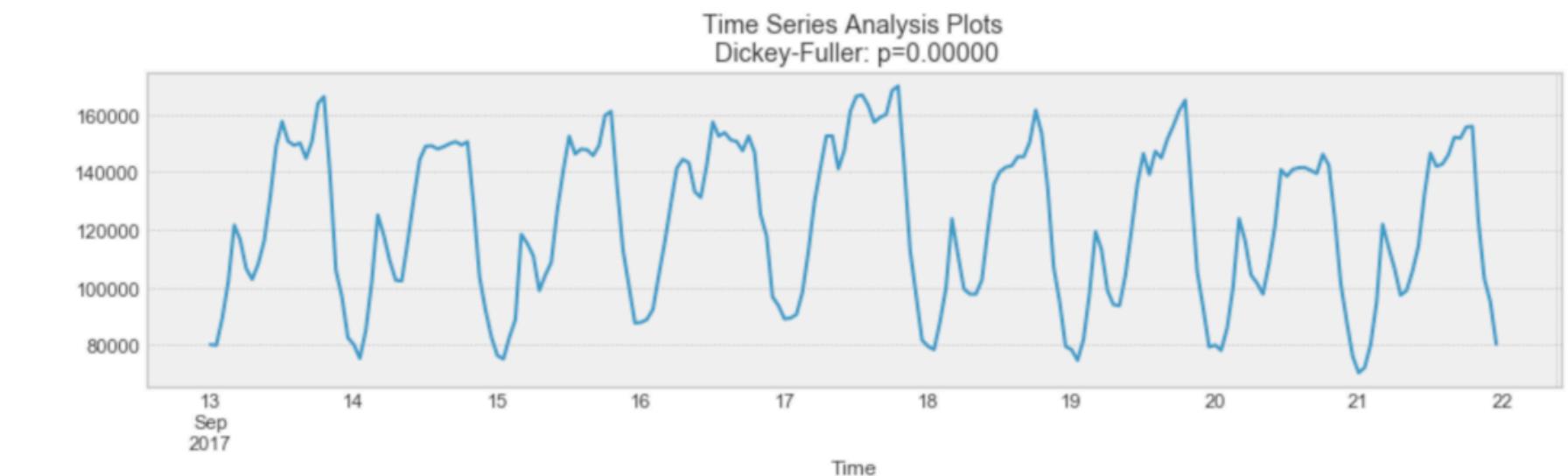
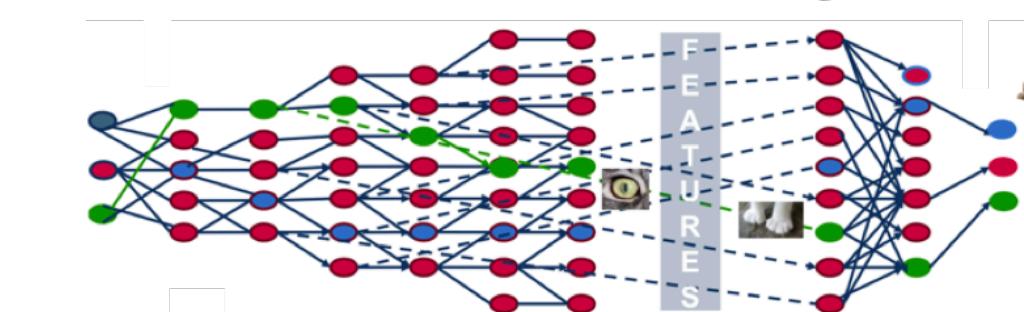


SPJ Queries

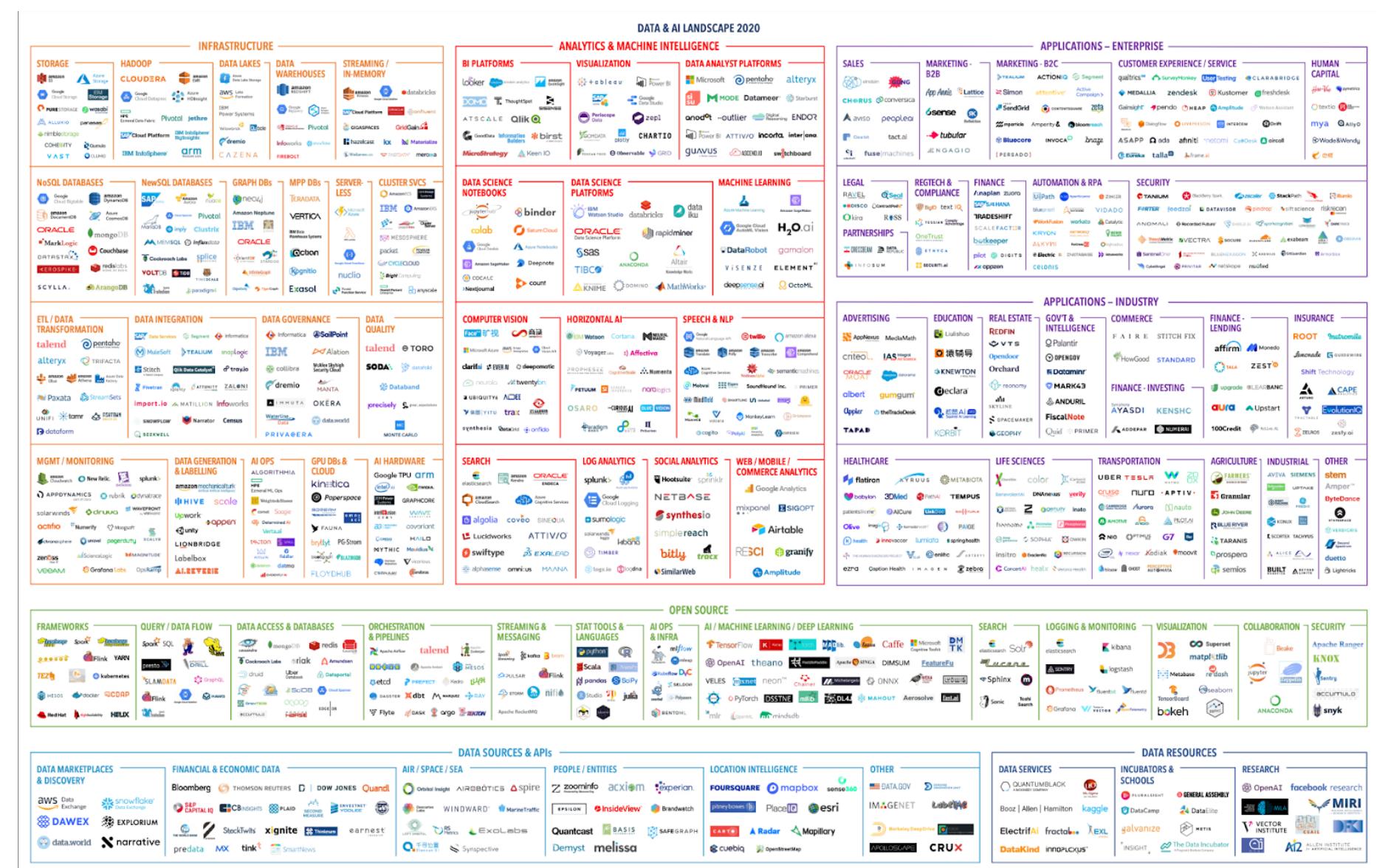
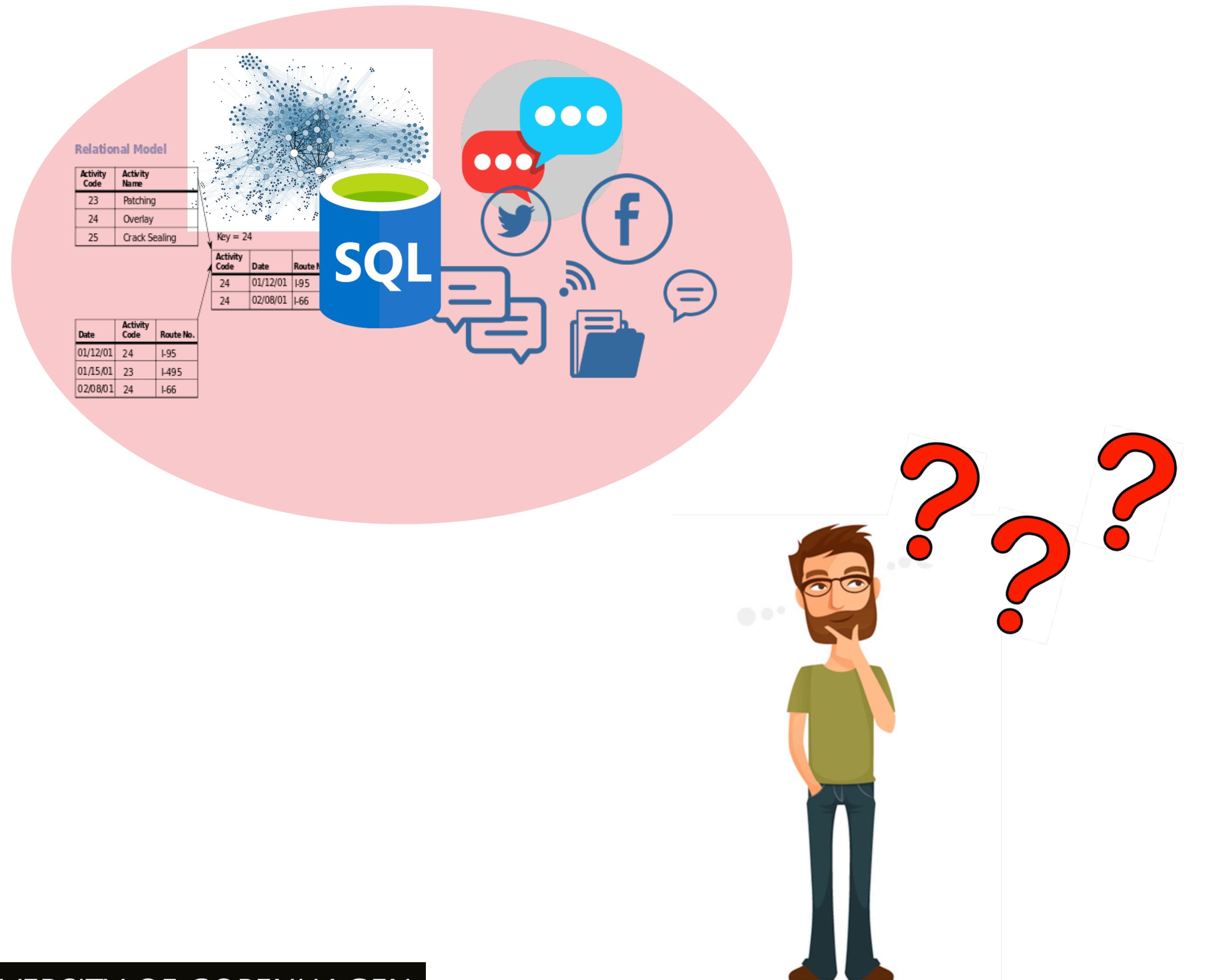
```
SELECT name  
FROM employee e  
INNER JOIN Person p  
ON p.firstname = e.name
```



Deep Learning



Users overwhelmed



Cross-platform Data Processing

use of multiple data processing platforms for query processing (single run or across several runs)

Cross-Platform Data Processing / Multi-engine Data Processing



Outline

- ◆ Motivation
- ◆ Cross-platform use cases
- ◆ Apache Wayang

Outline

- ◆ Motivation
- ◆ **Cross-platform use cases**
- ◆ Apache Wayang

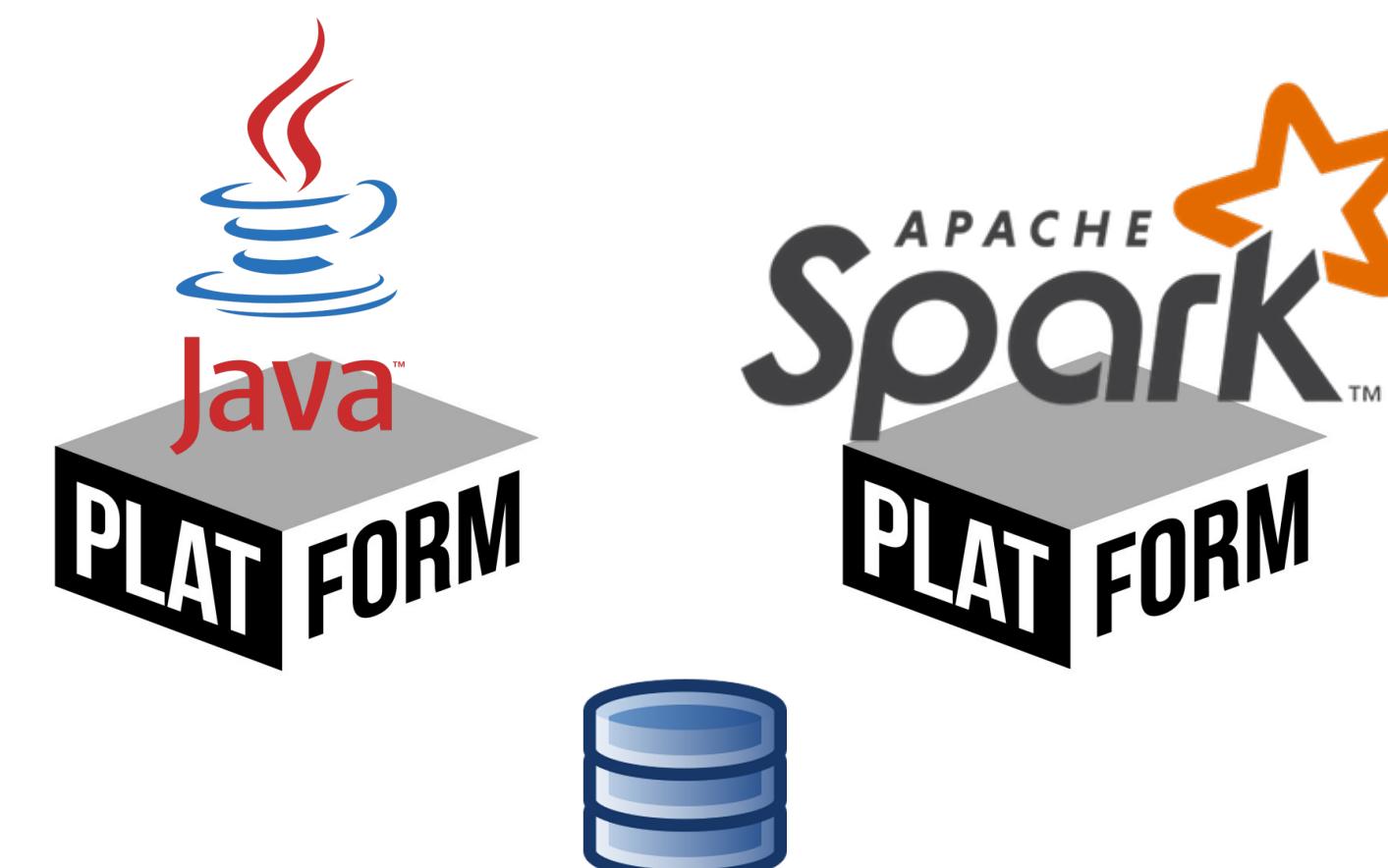
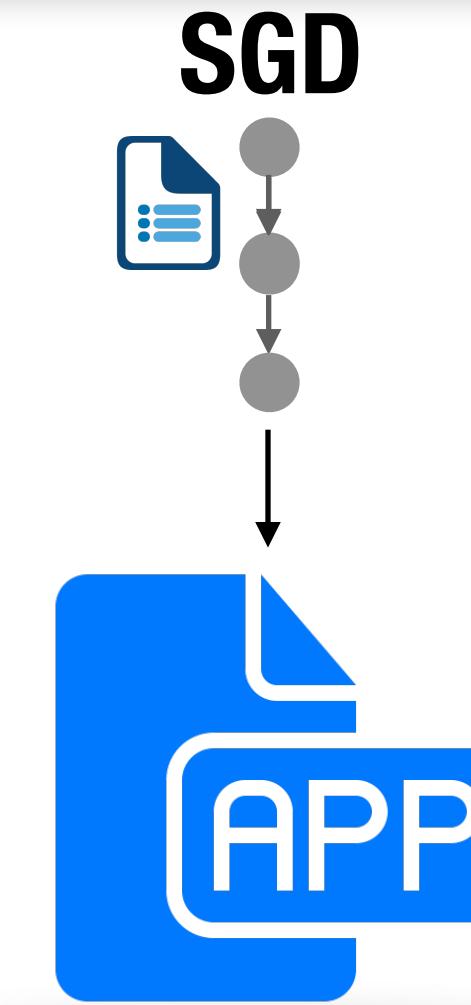
Decoupled single-platform

using ***any single*** data processing platform to process a query



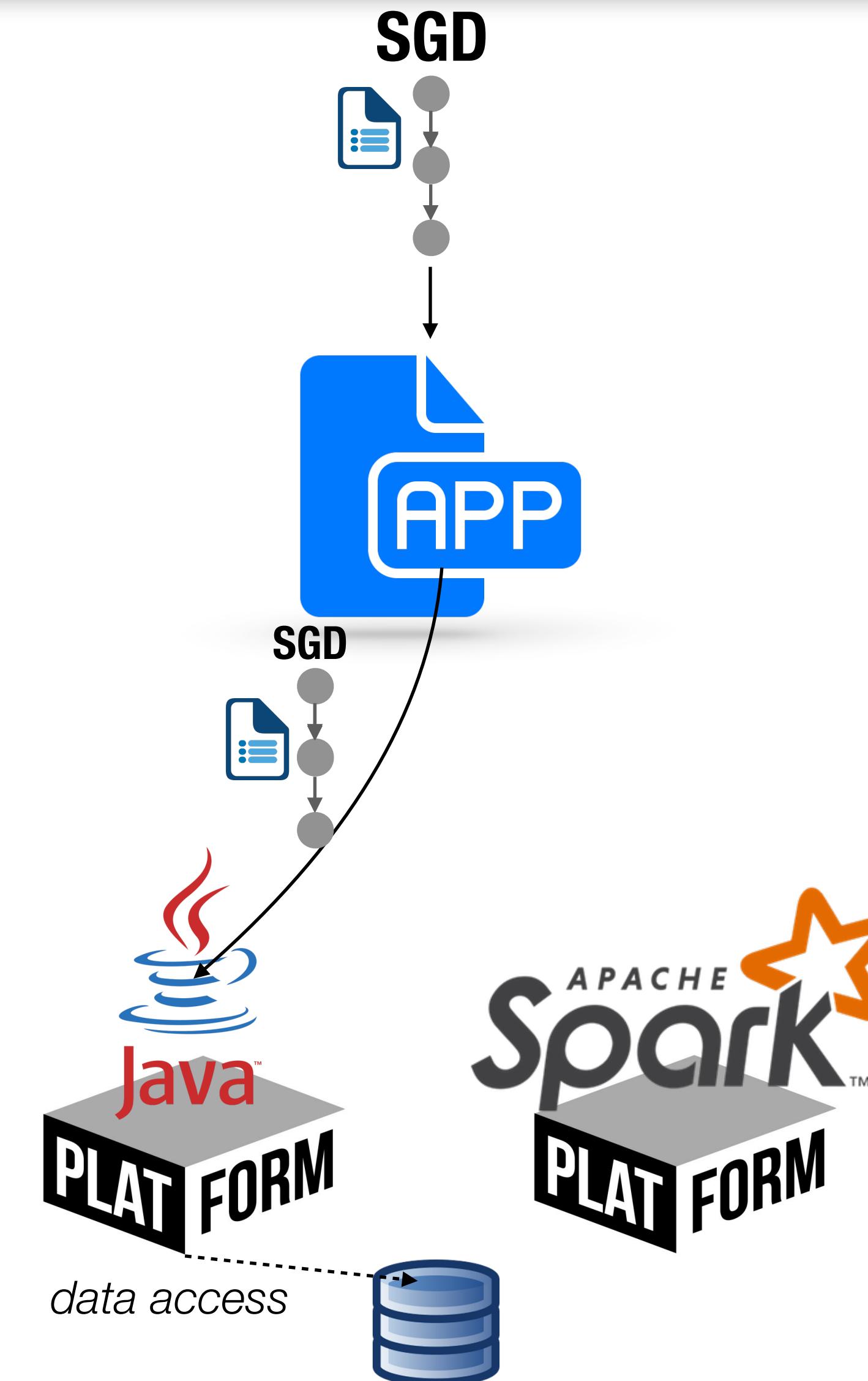
Decoupled single-platform

using ***any single*** data processing platform to process a query



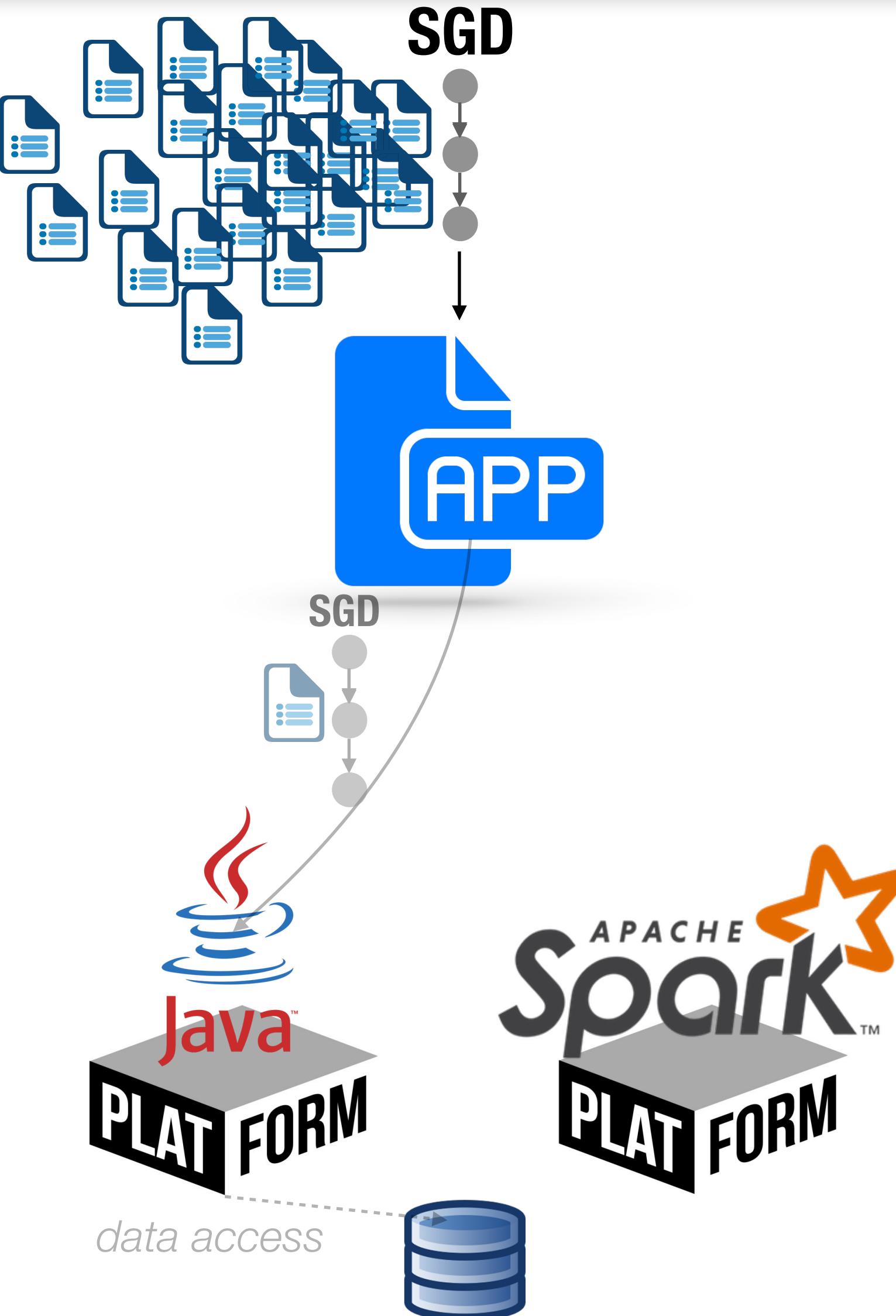
Decoupled single-platform

using ***any single*** data processing platform to process a query



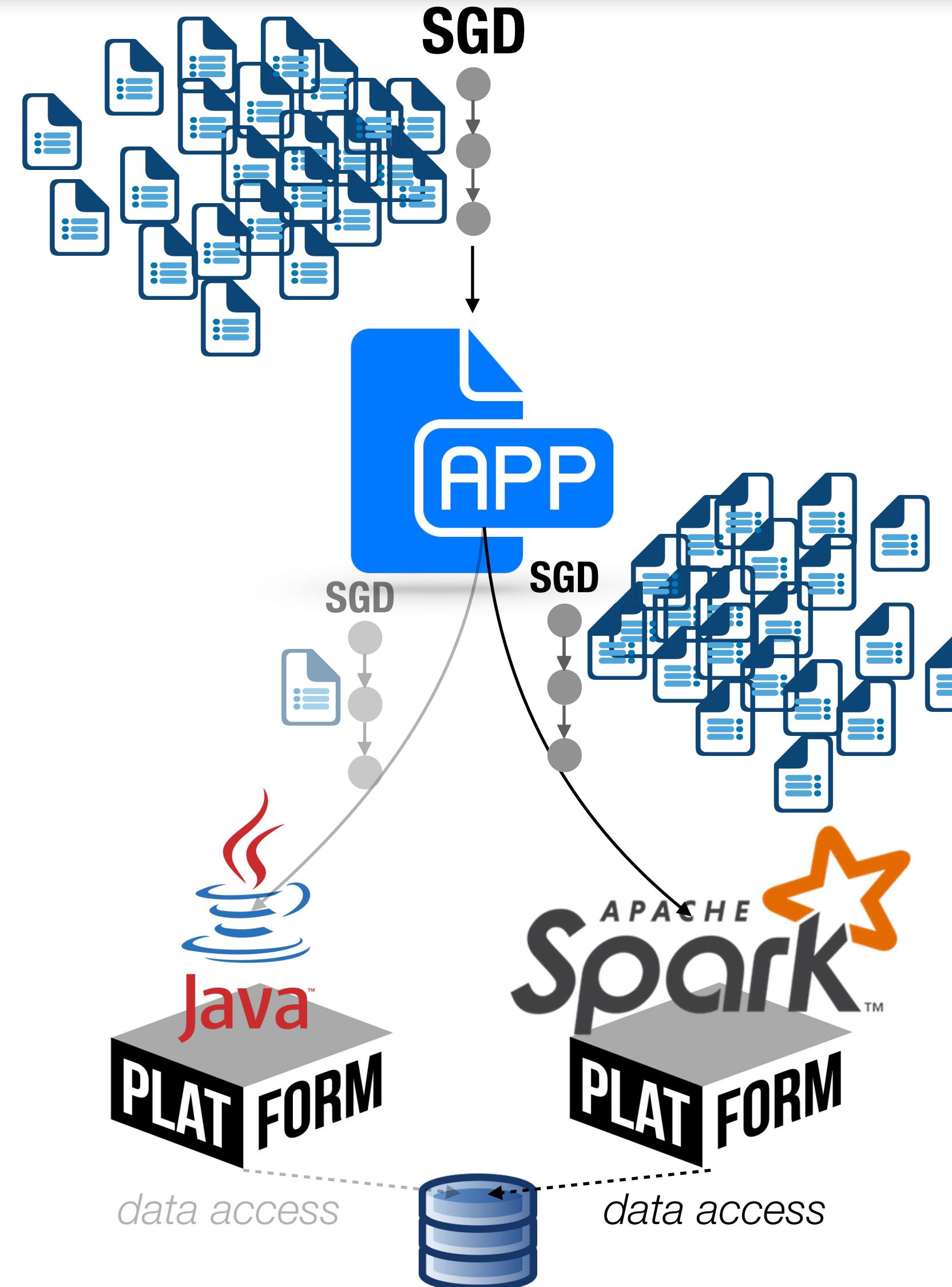
Decoupled single-platform

using **any single** data processing platform to process a query



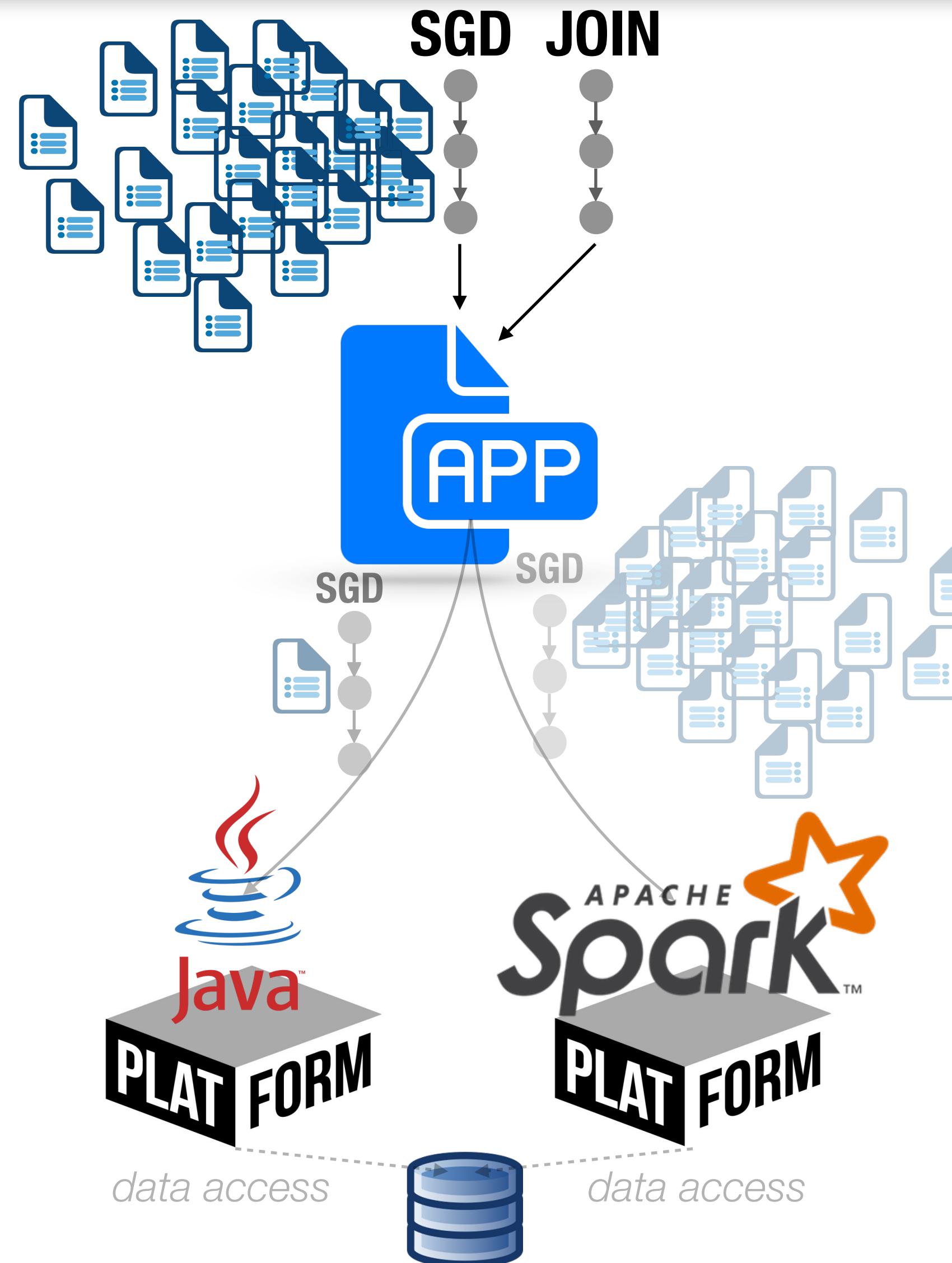
Decoupled single-platform

using **any single** data processing platform to process a query



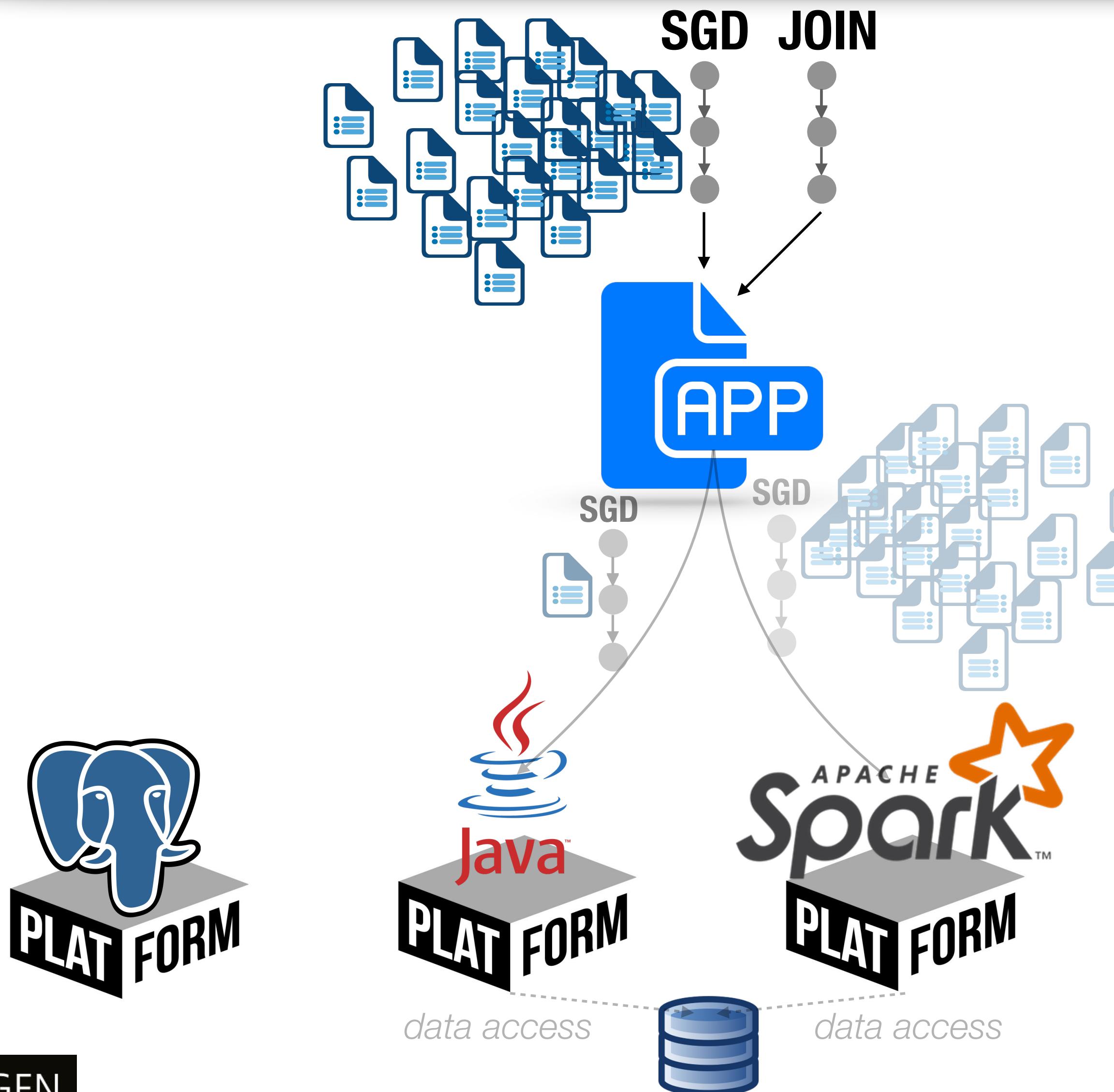
Decoupled single-platform

using *any single* data processing platform to process a query



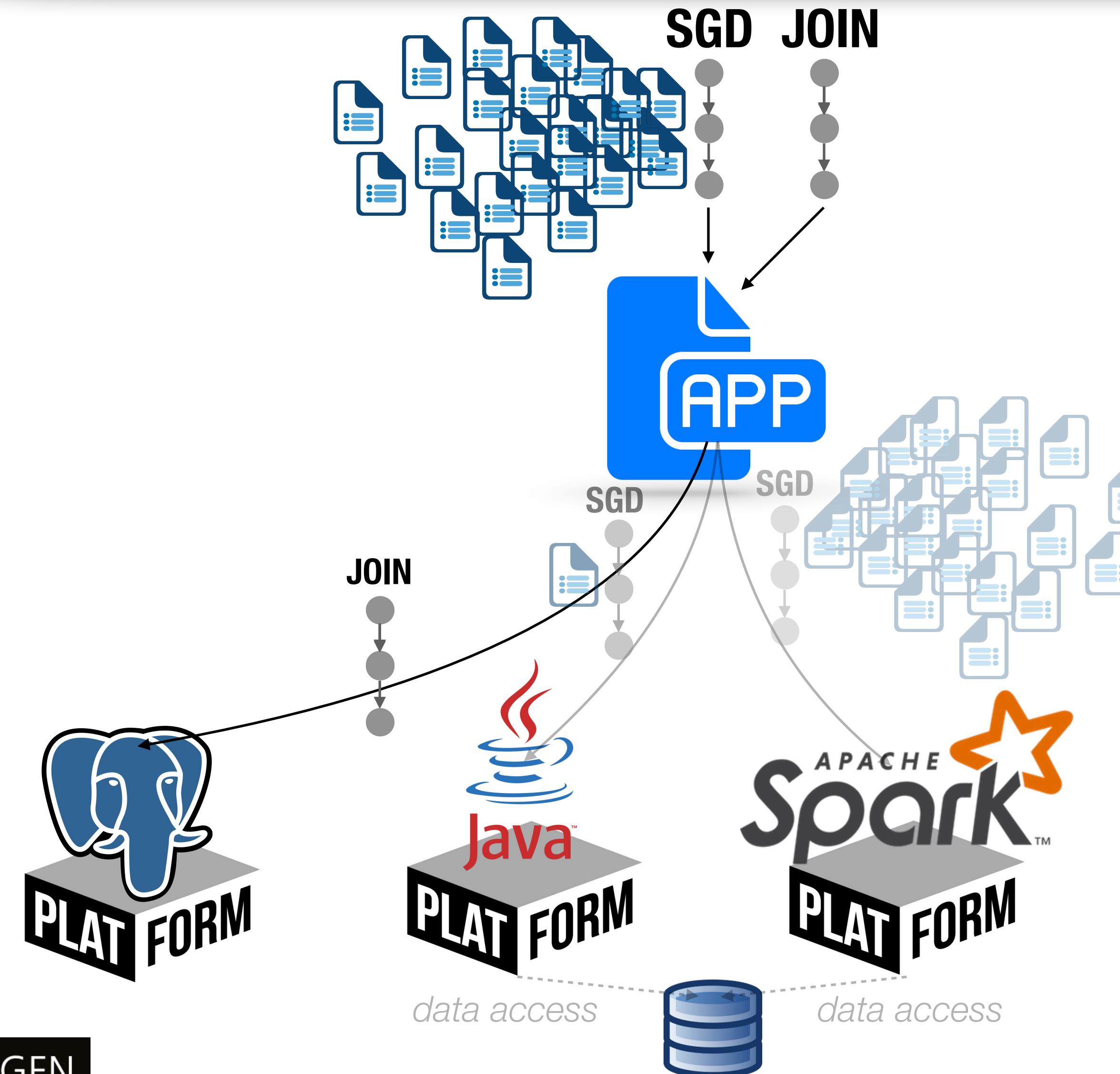
Decoupled single-platform

using ***any single*** data processing platform to process a query



Decoupled single-platform

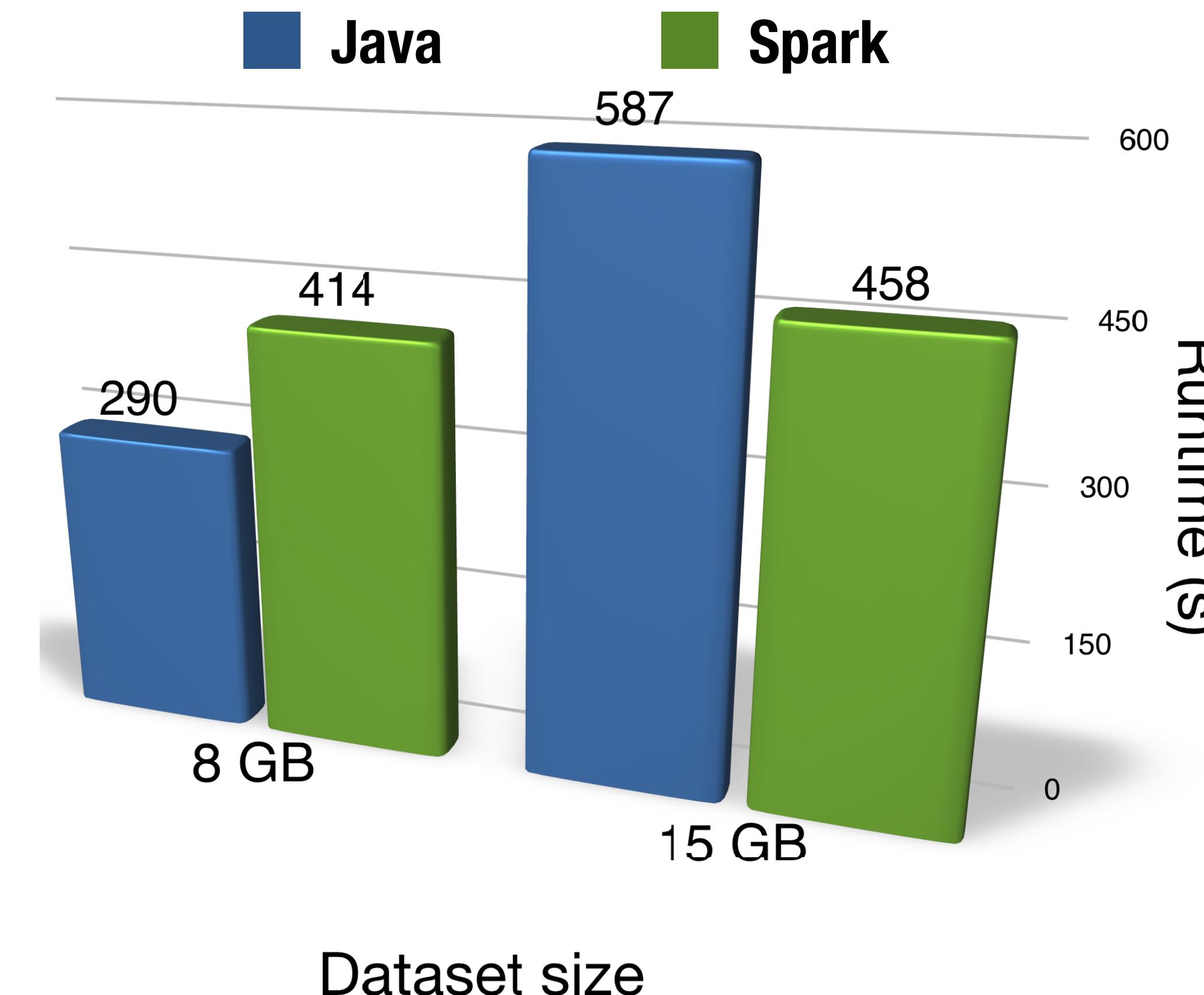
using *any single* data processing platform to process a query



Decoupled single-platform

using *any single* data processing platform to process a query

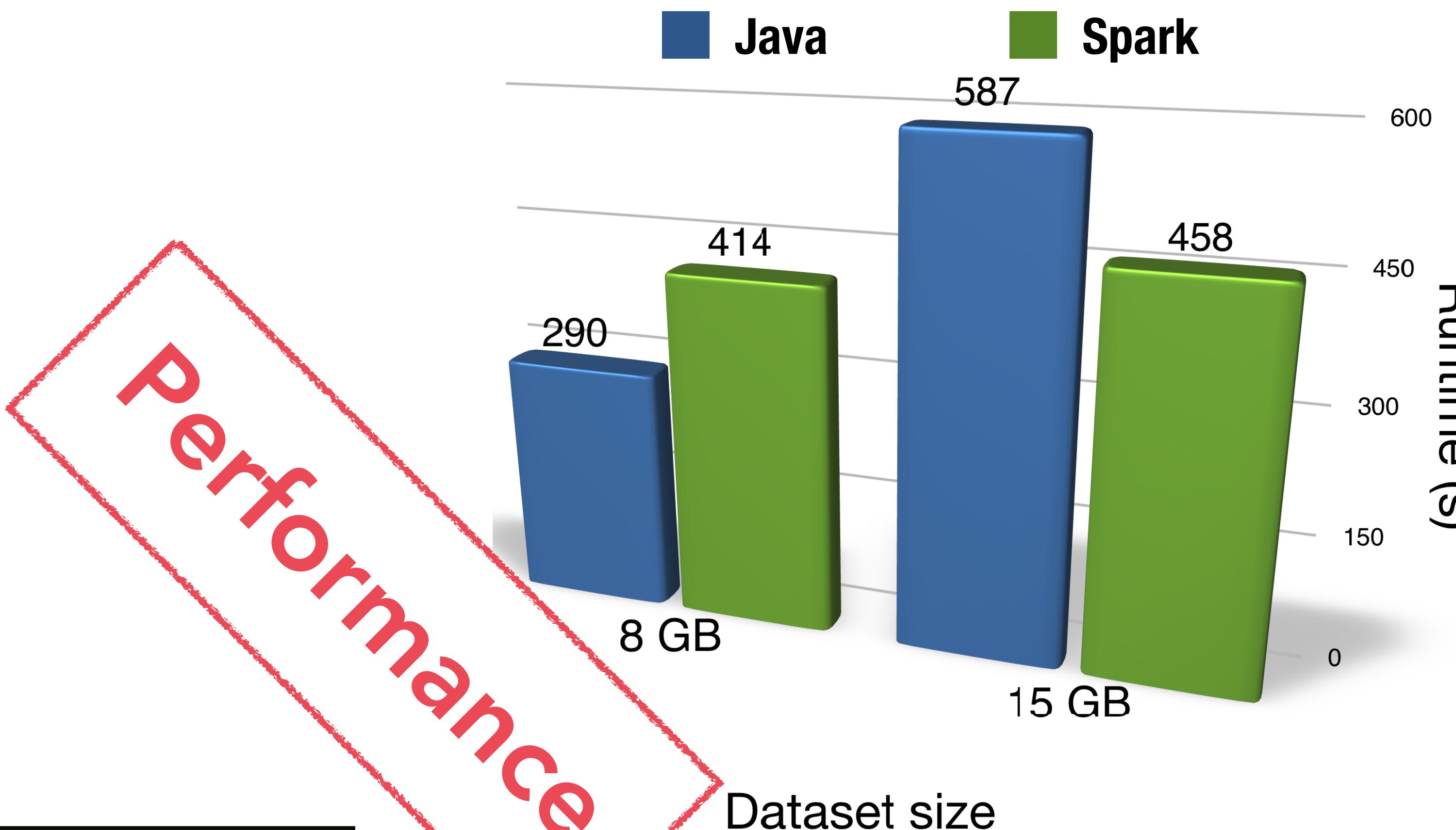
SGD



Decoupled single-platform

using **any single** data processing platform to process a query

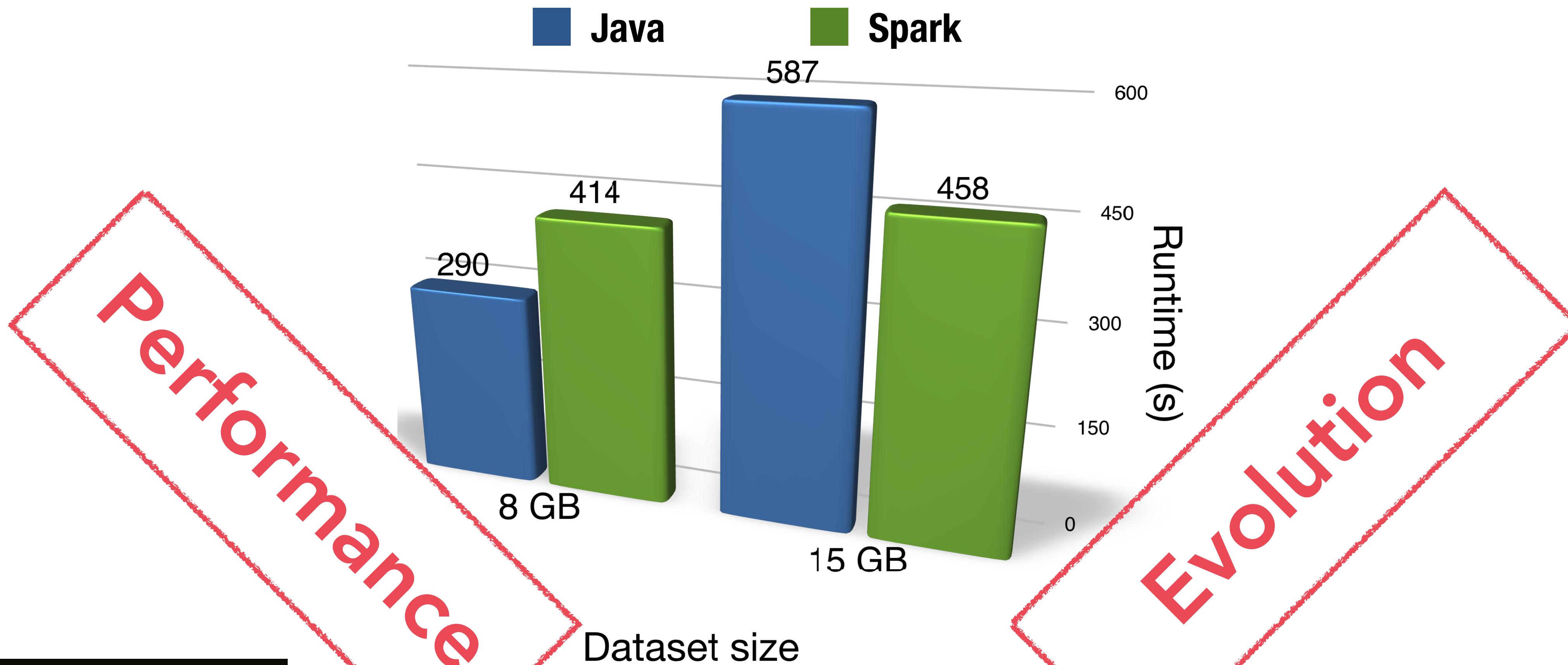
SGD



Decoupled single-platform

using *any single* data processing platform to process a query

SGD



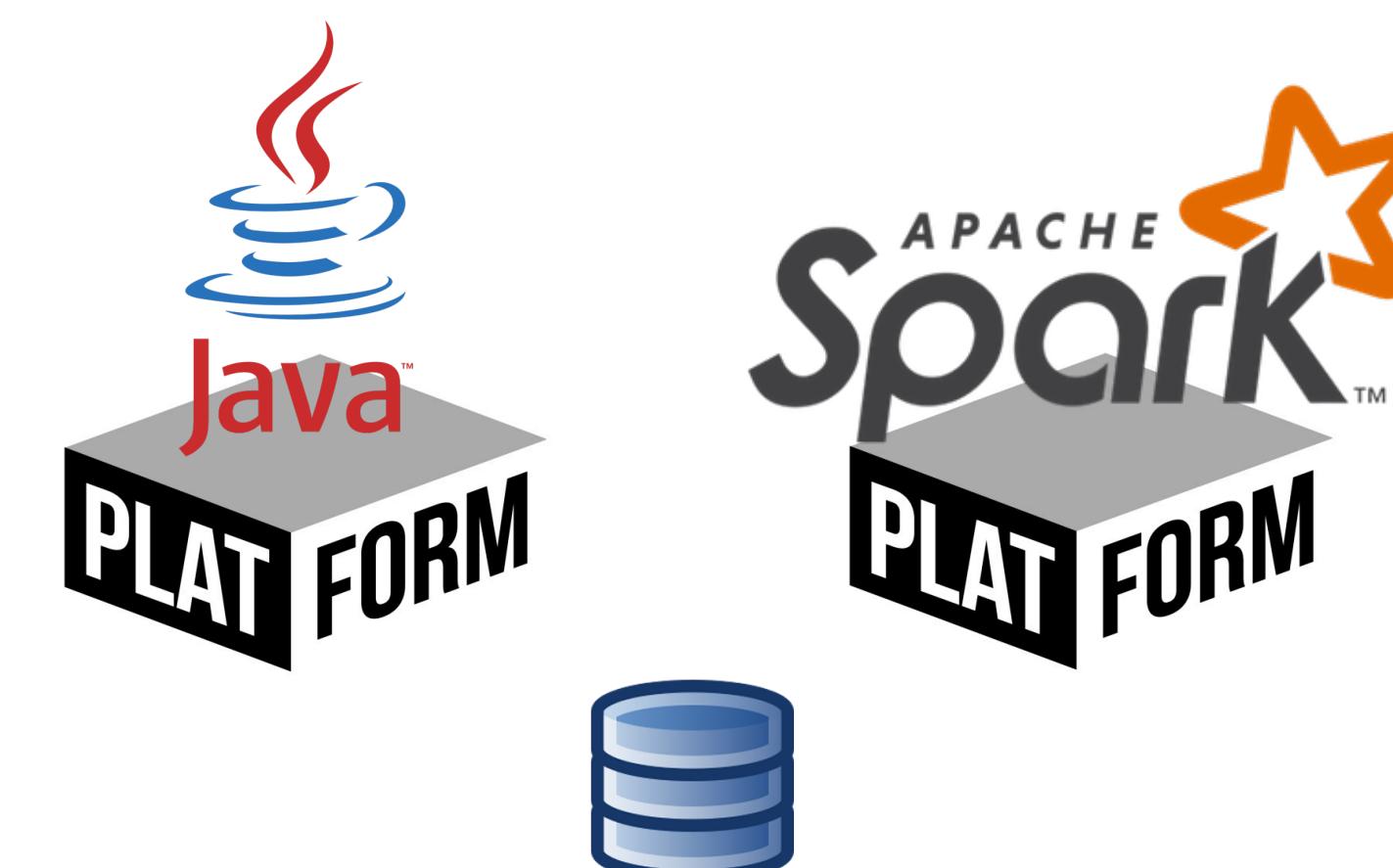
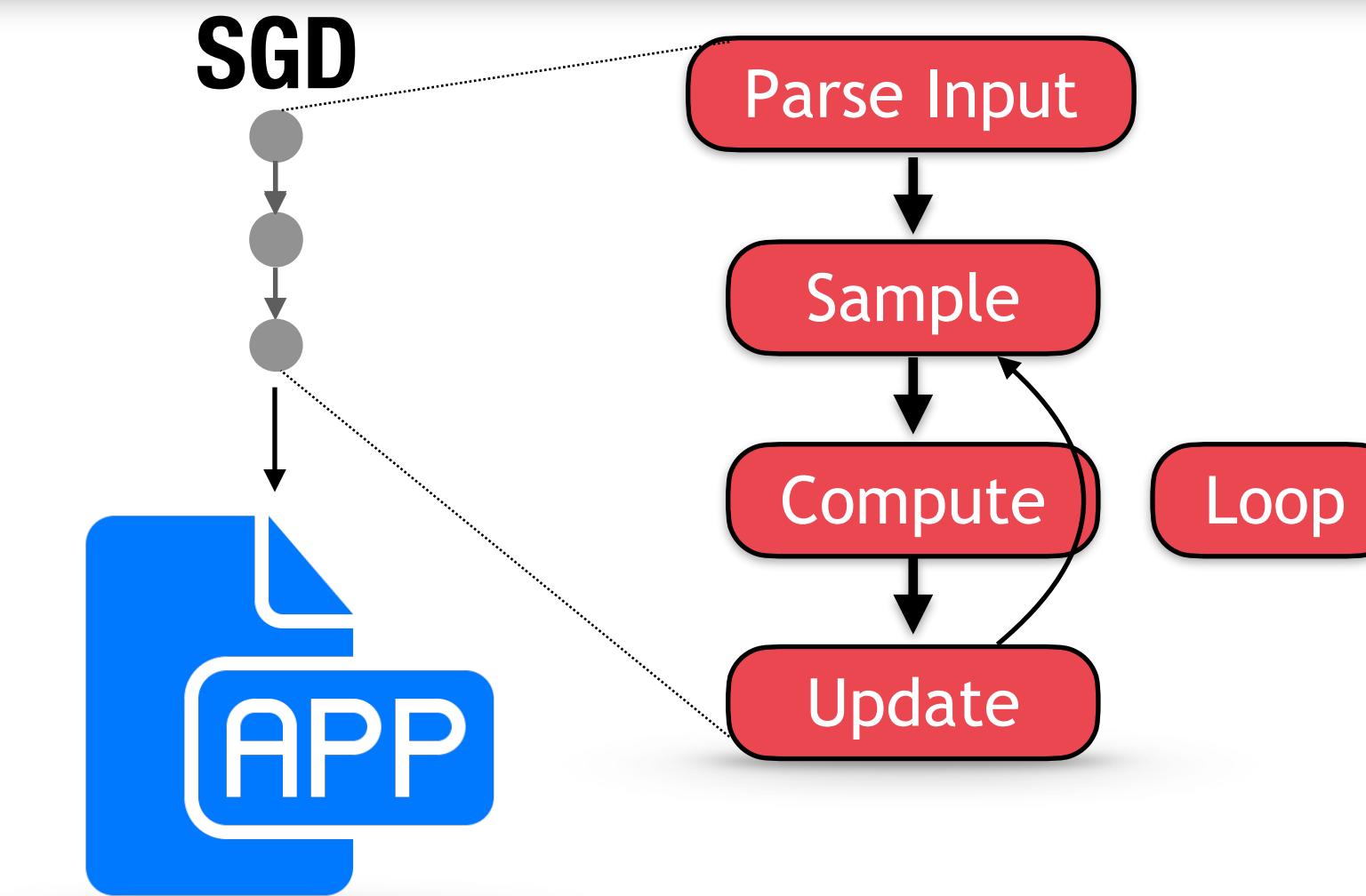
Opportunistic cross-platform

using **several** data processing platforms to reduce the cost of a query



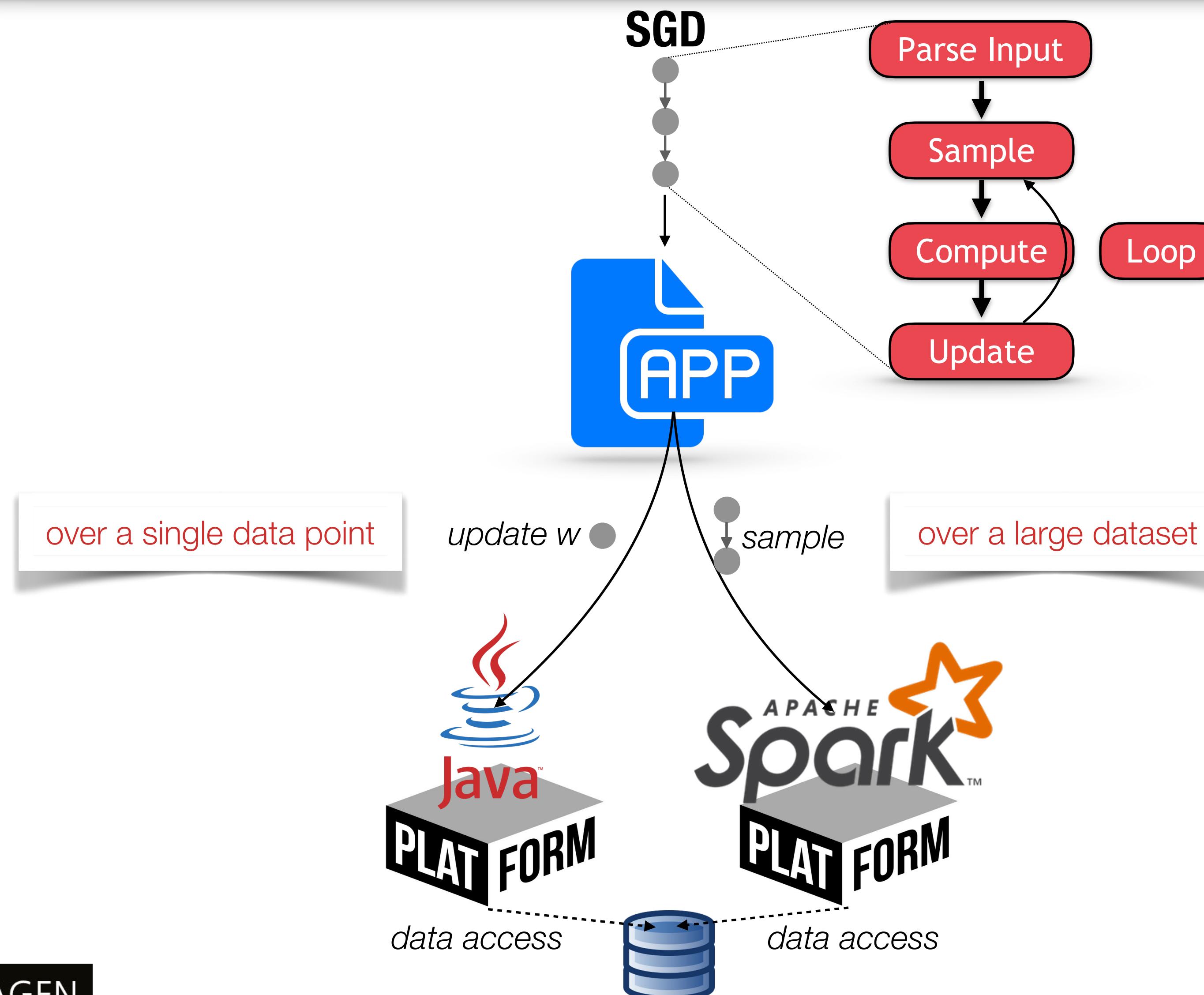
Opportunistic cross-platform

using **several** data processing platforms to reduce the cost of a query



Opportunistic cross-platform

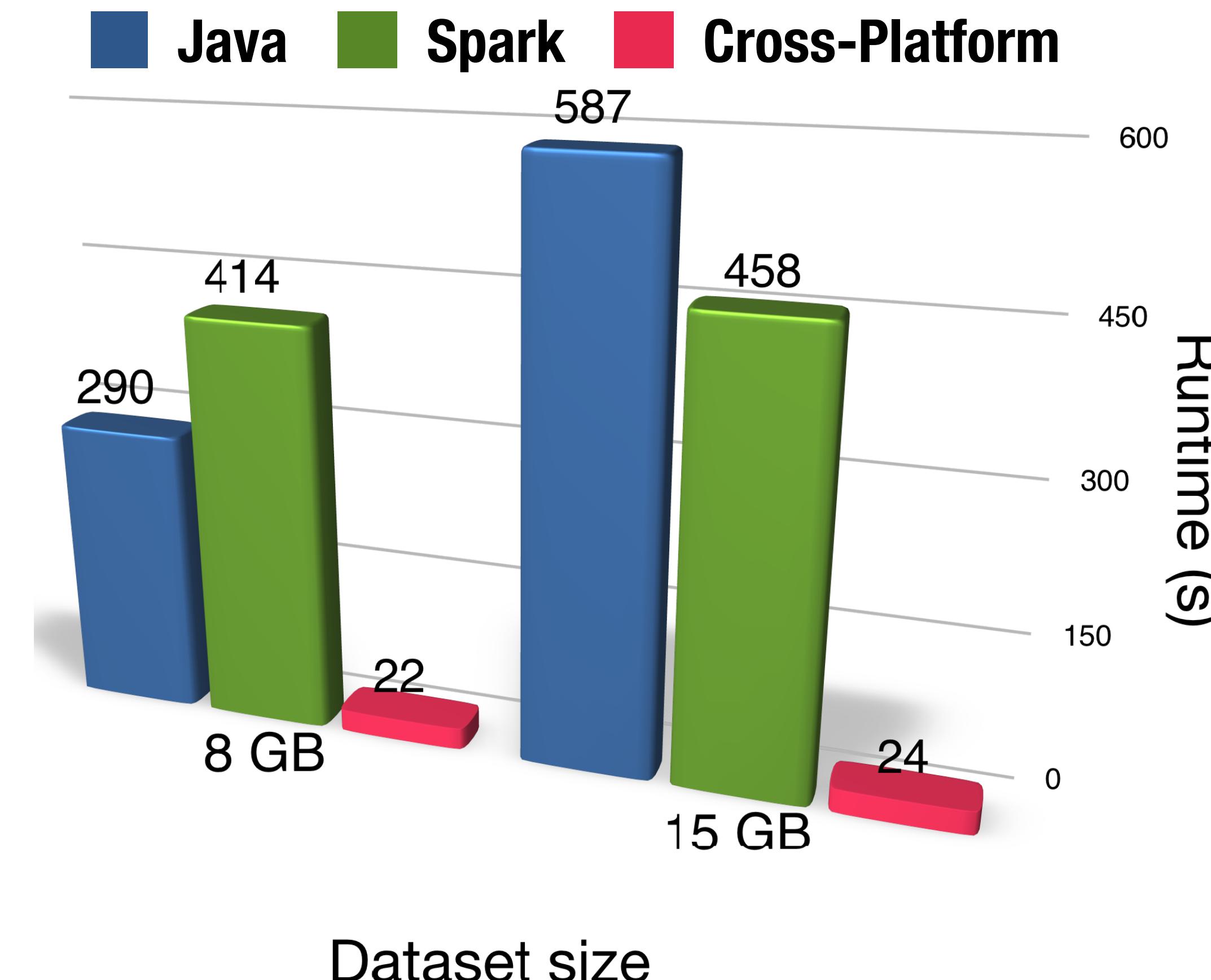
using **several** data processing platforms to reduce the cost of a query



Opportunistic cross-platform

using **several** data processing platforms to reduce the cost of a query

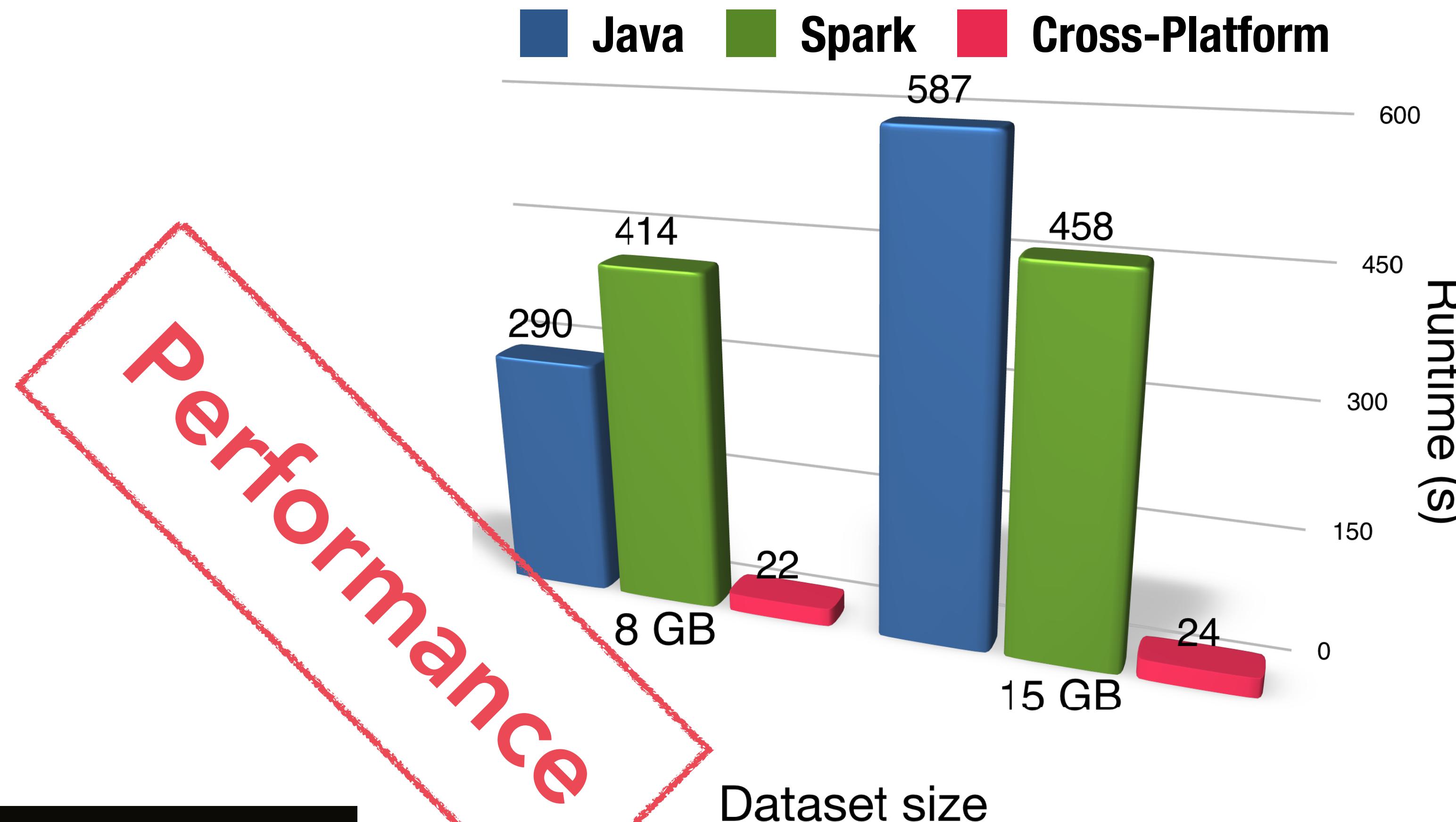
SGD



Opportunistic cross-platform

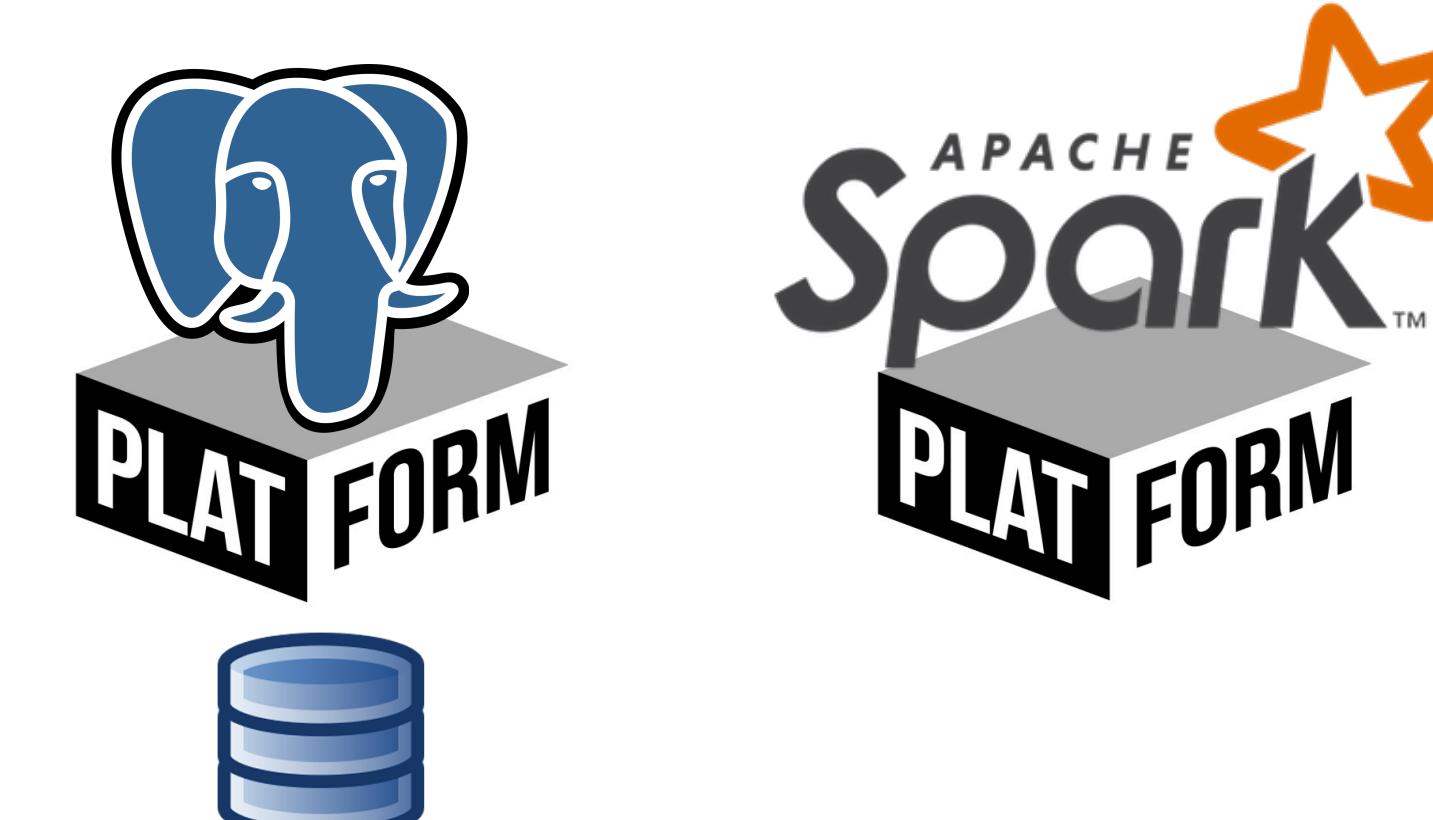
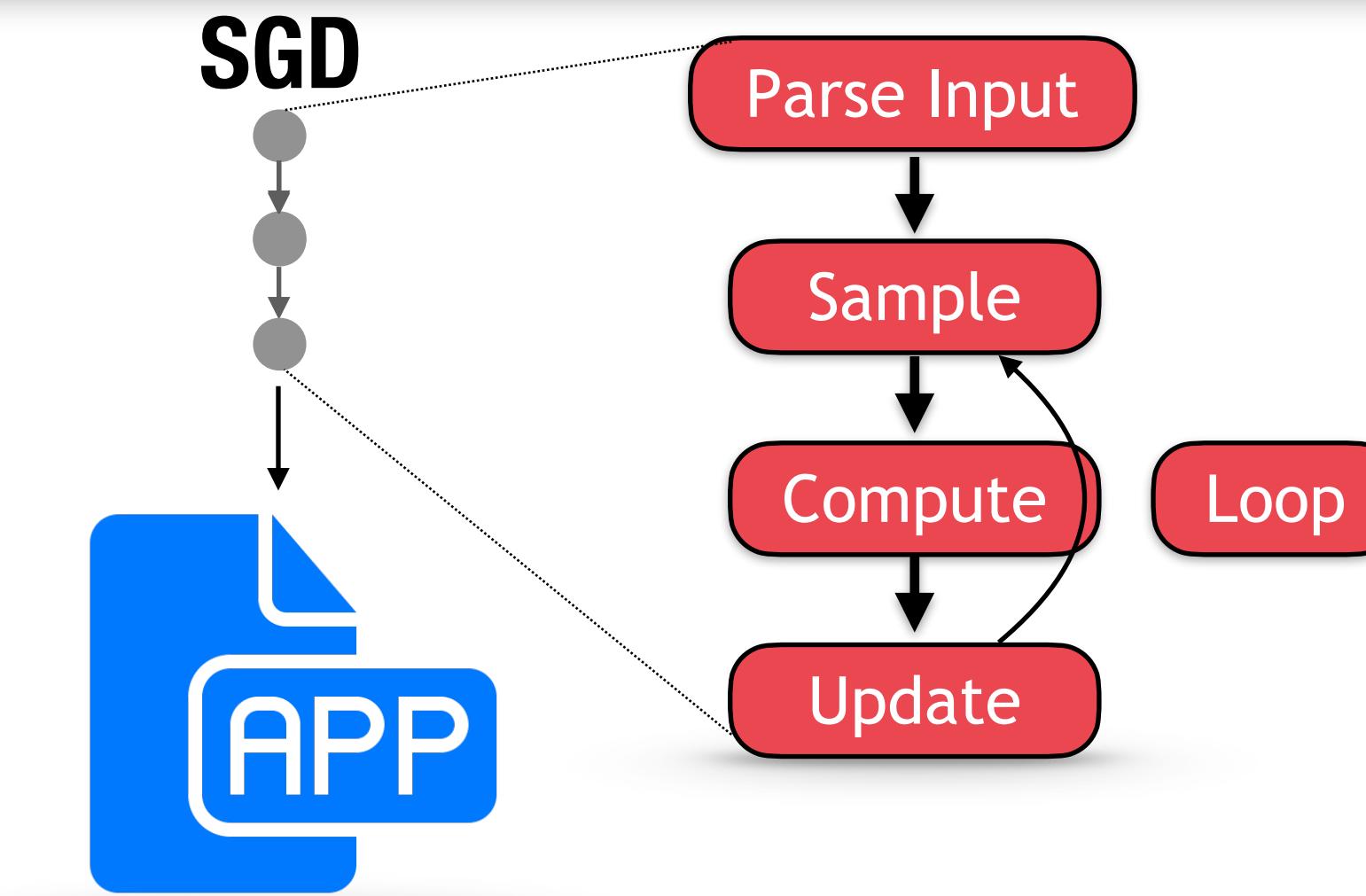
using **several** data processing platforms to reduce the cost of a query

SGD



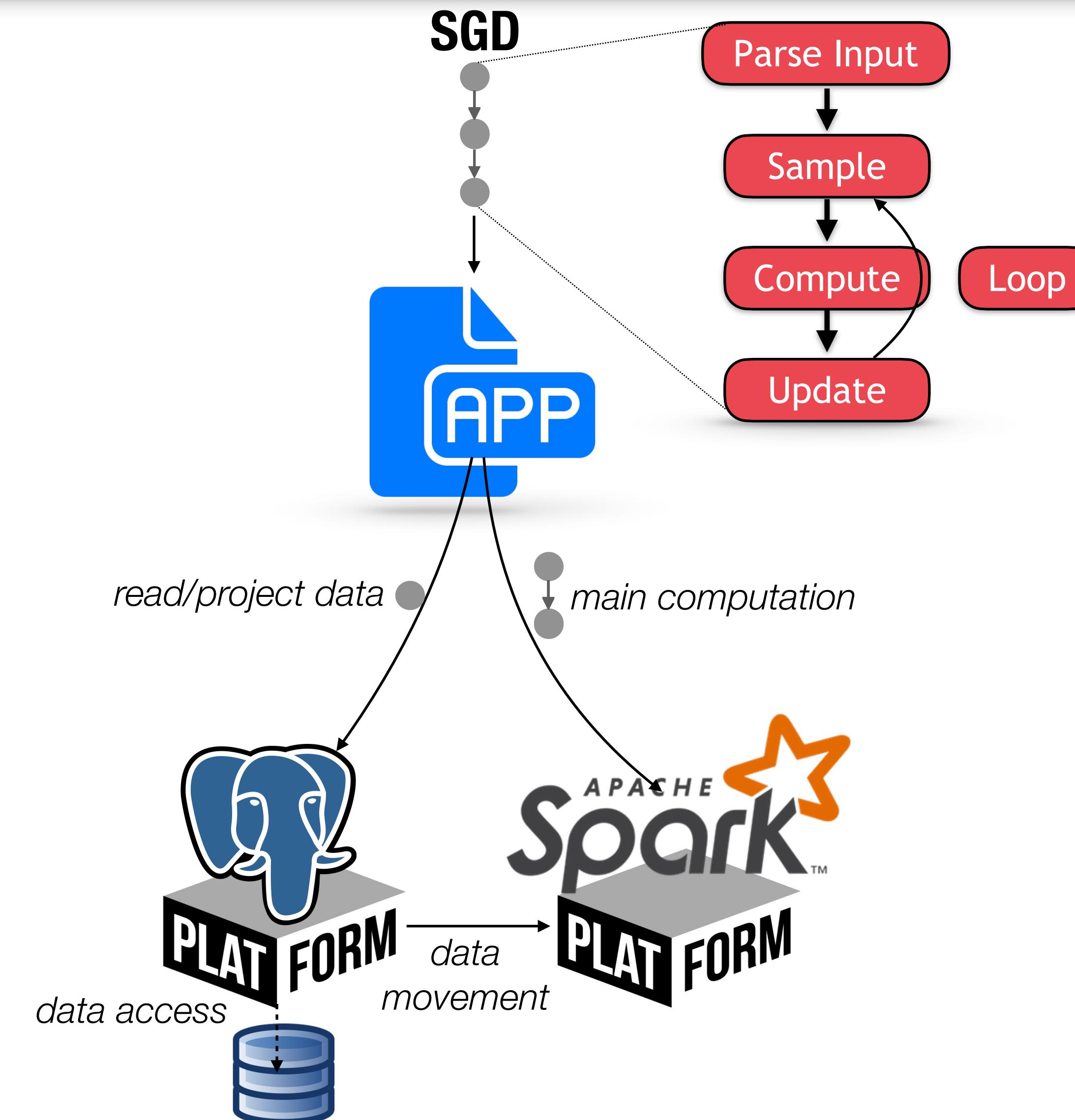
Mandatory cross-platform

using **several** data processing platforms to be able to process a query



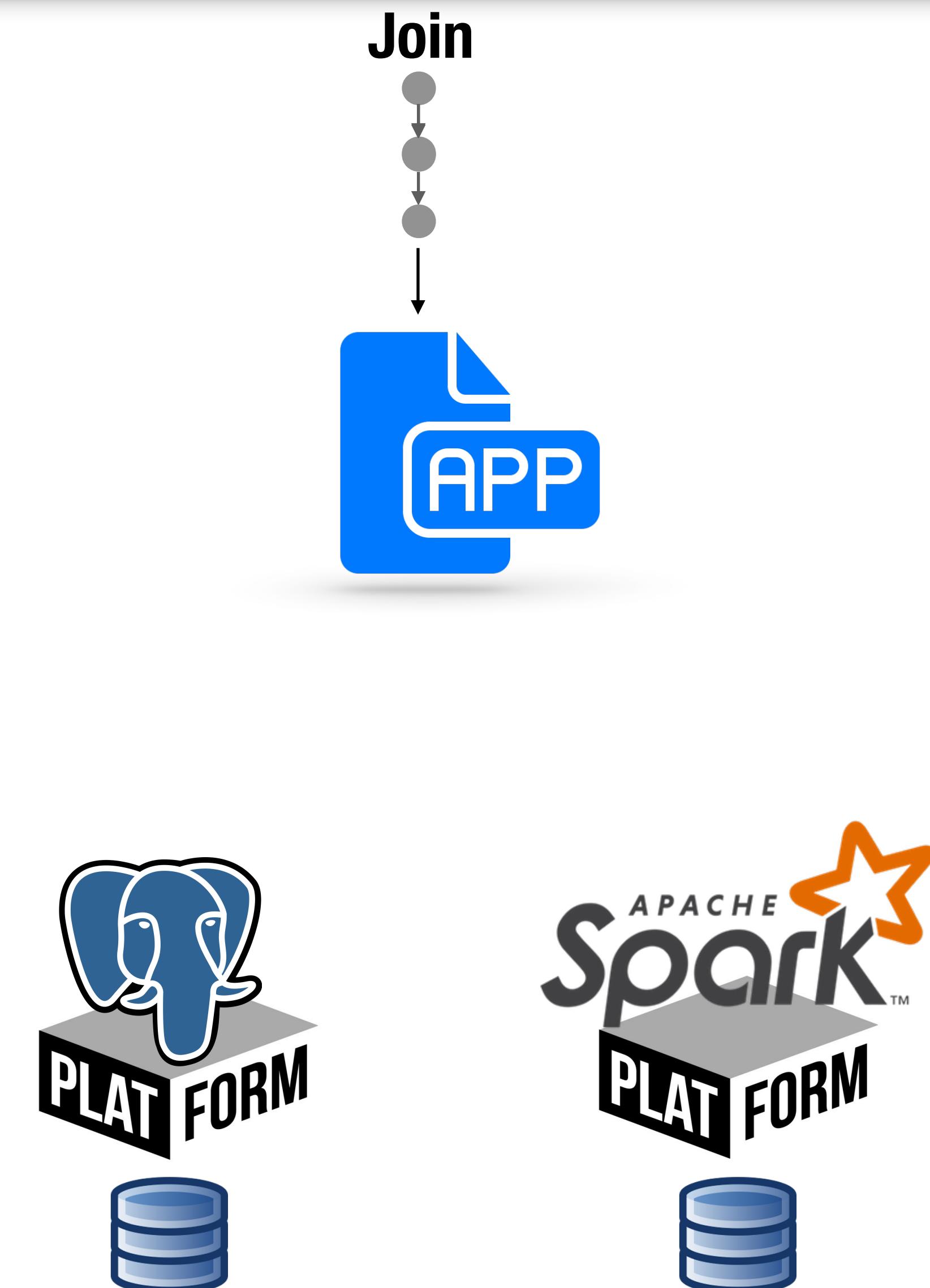
Mandatory cross-platform

using **several** data processing platforms to be able to process a query



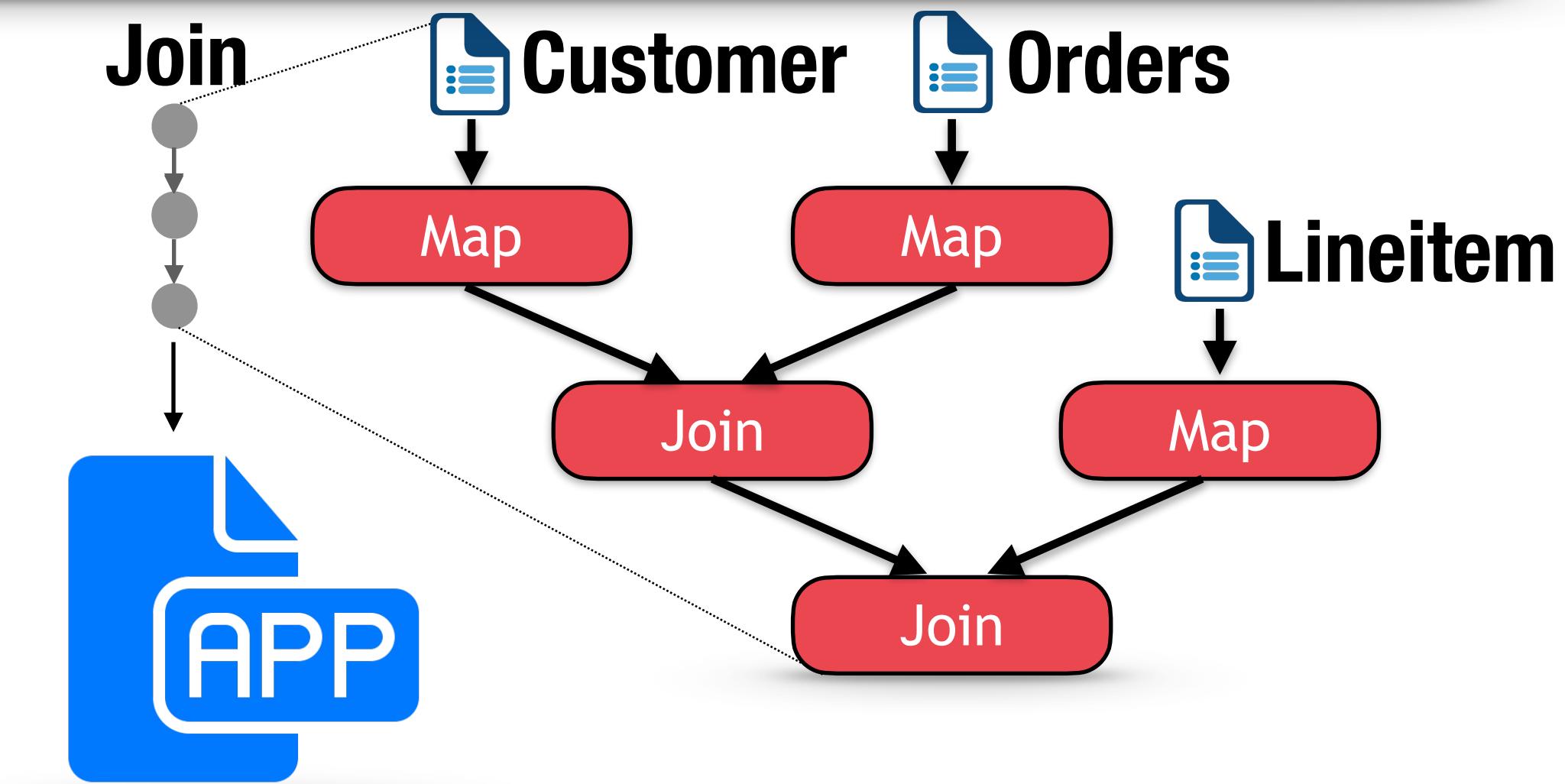
Polystore

using **several** data processing platforms because data is spread



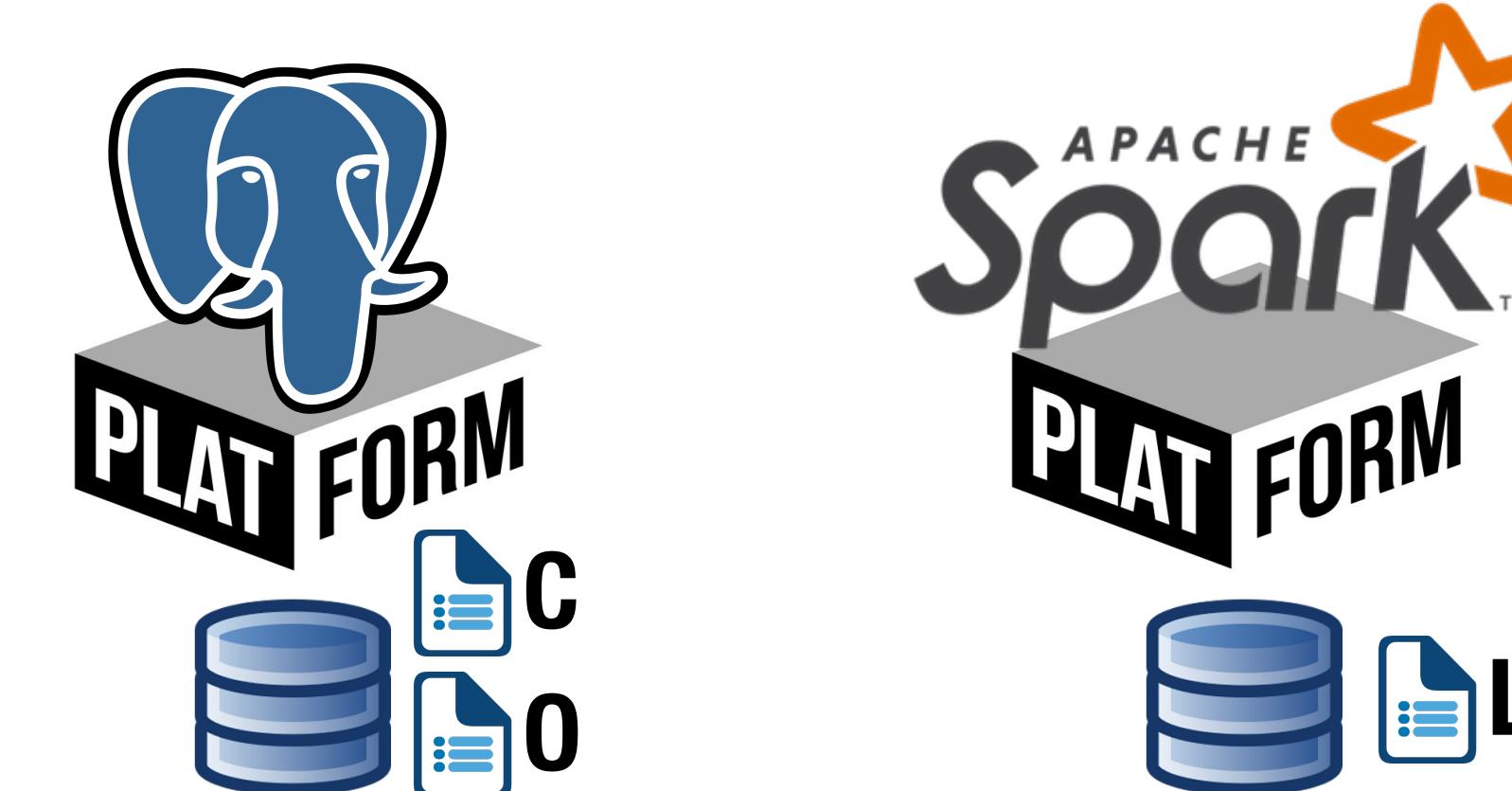
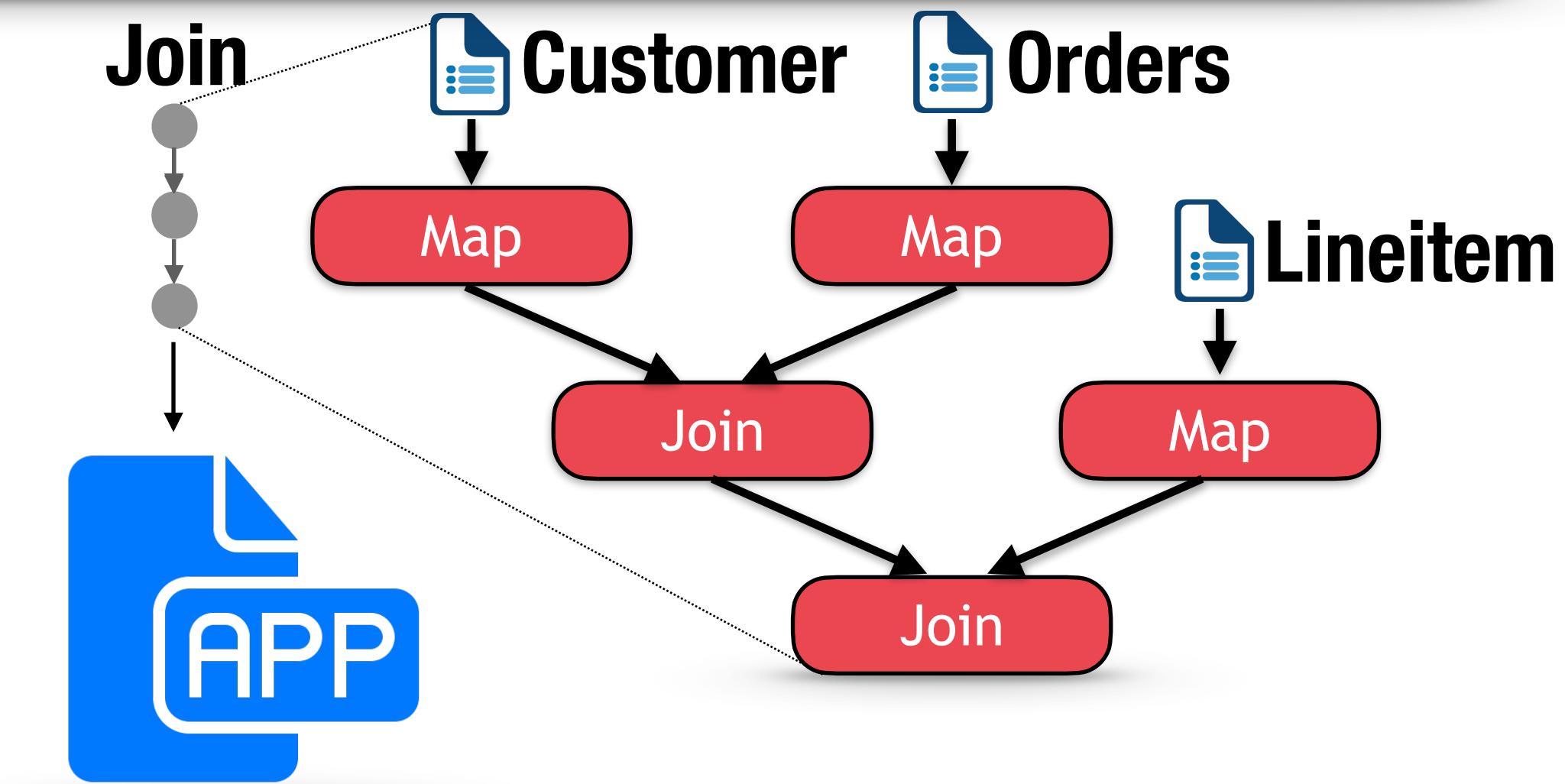
Polystore

using **several** data processing platforms because data is spread



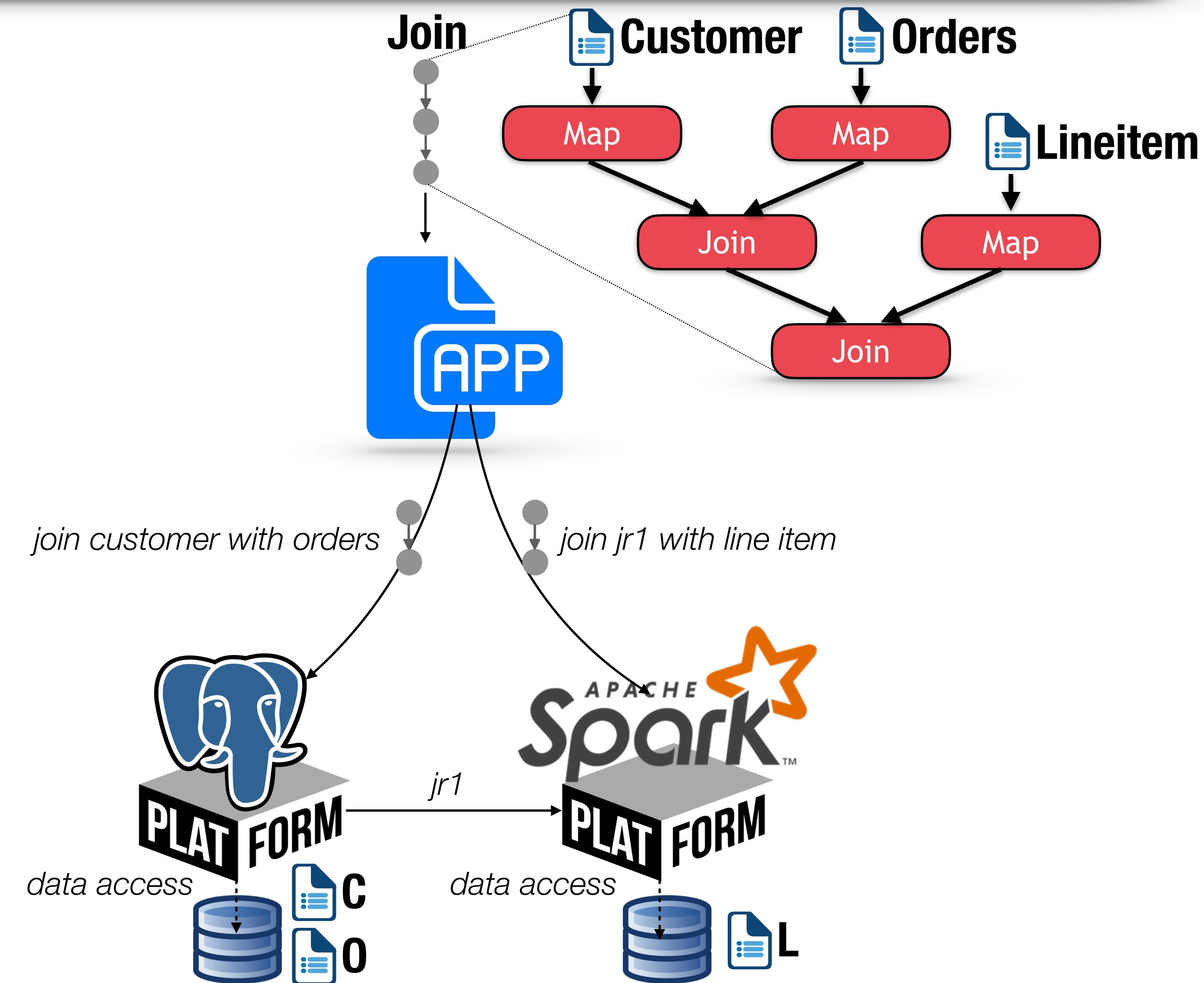
Polystore

using **several** data processing platforms because data is spread



Polystore

using **several** data processing platforms because data is spread



Outline

- ❖ Motivation
- ❖ Cross-platform use cases
- ❖ **Apache Wayang**

Closer look into Apache Wayang



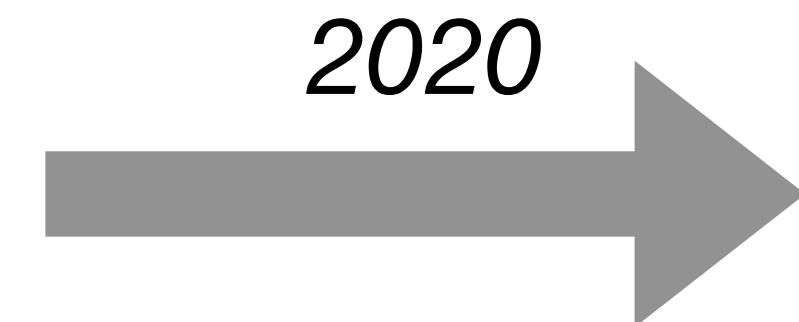
Publications



RHEEM: Enabling Cross-Platform Data Processing - May The Big Data Be With You! -. PVLDB 11(11): 1414-1427 (2018)

RHEEMix in the data jungle: a cost-based optimizer for cross-platform systems. VLDB J. 29(6): 1287-1310 (2020)

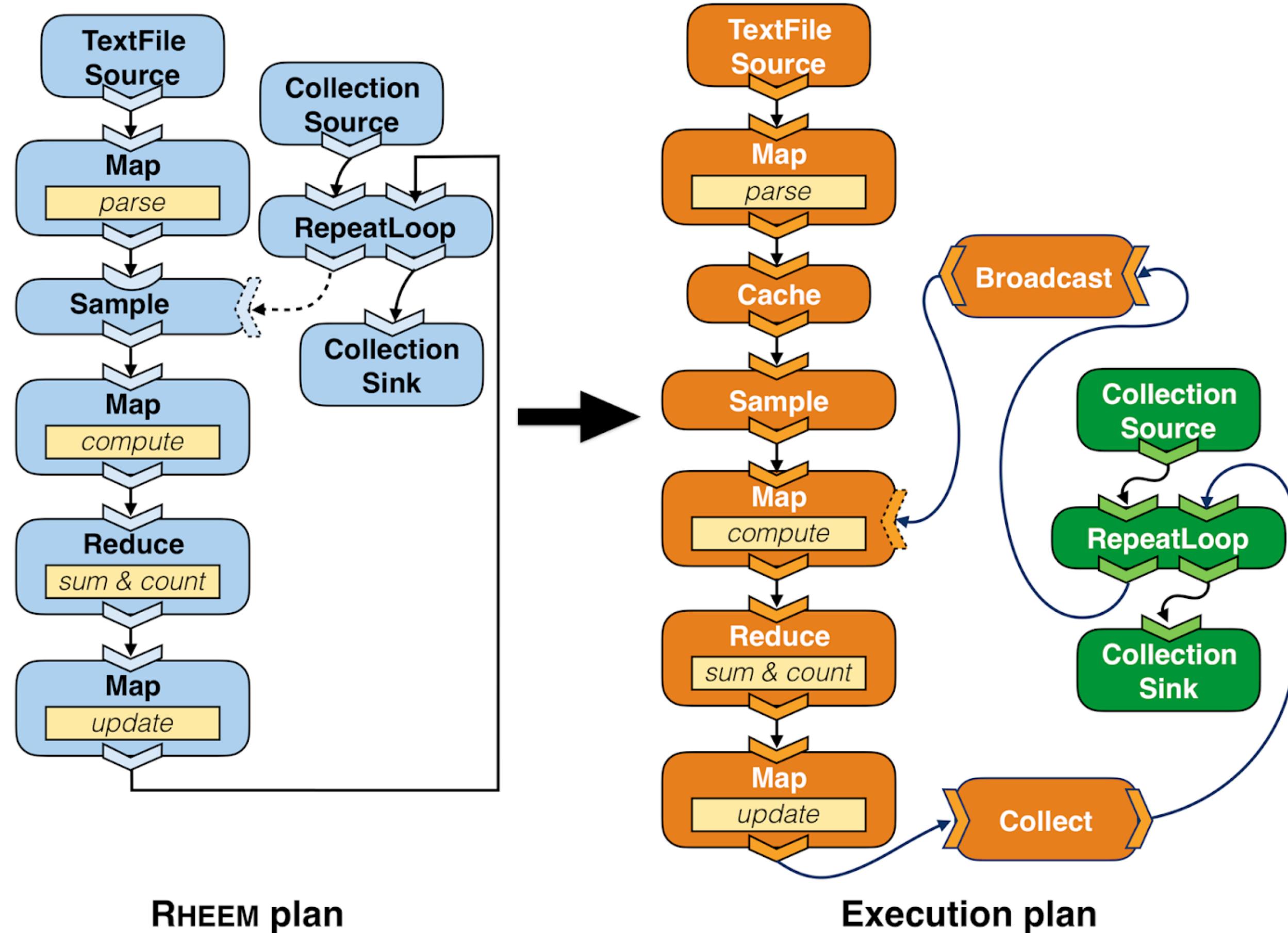
ML-based Cross-Platform Query Optimization. ICDE 2020: 1489-1500



<https://wayang.apache.org/>

Goal of Wayang

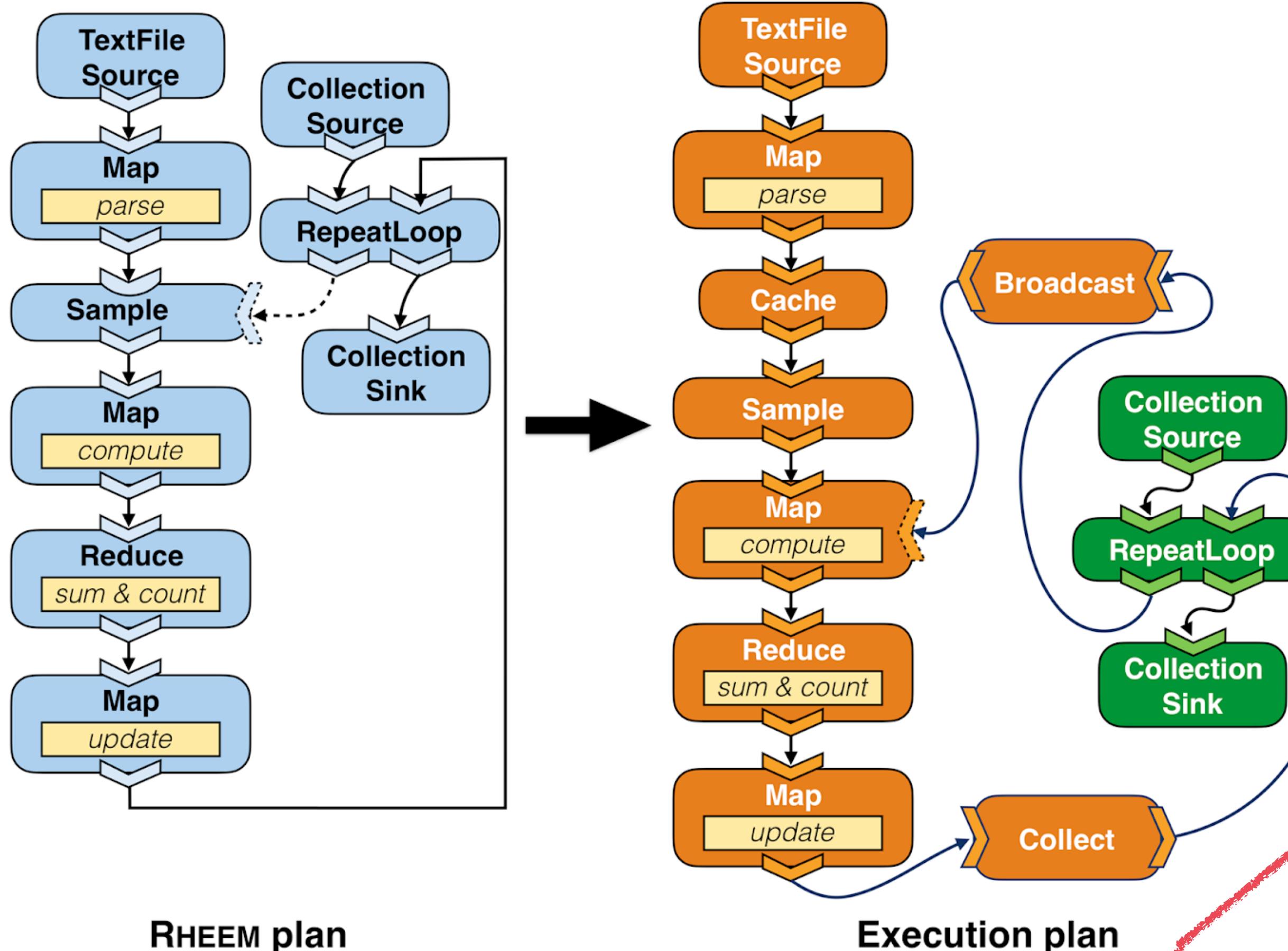
■ RHEEM operator ■ Spark execution operator ■ JavaStreams execution operator
➤ Input/Output ■ UDF → Data flow ⤵ Broadcast data flow



<https://wayang.apache.org/>

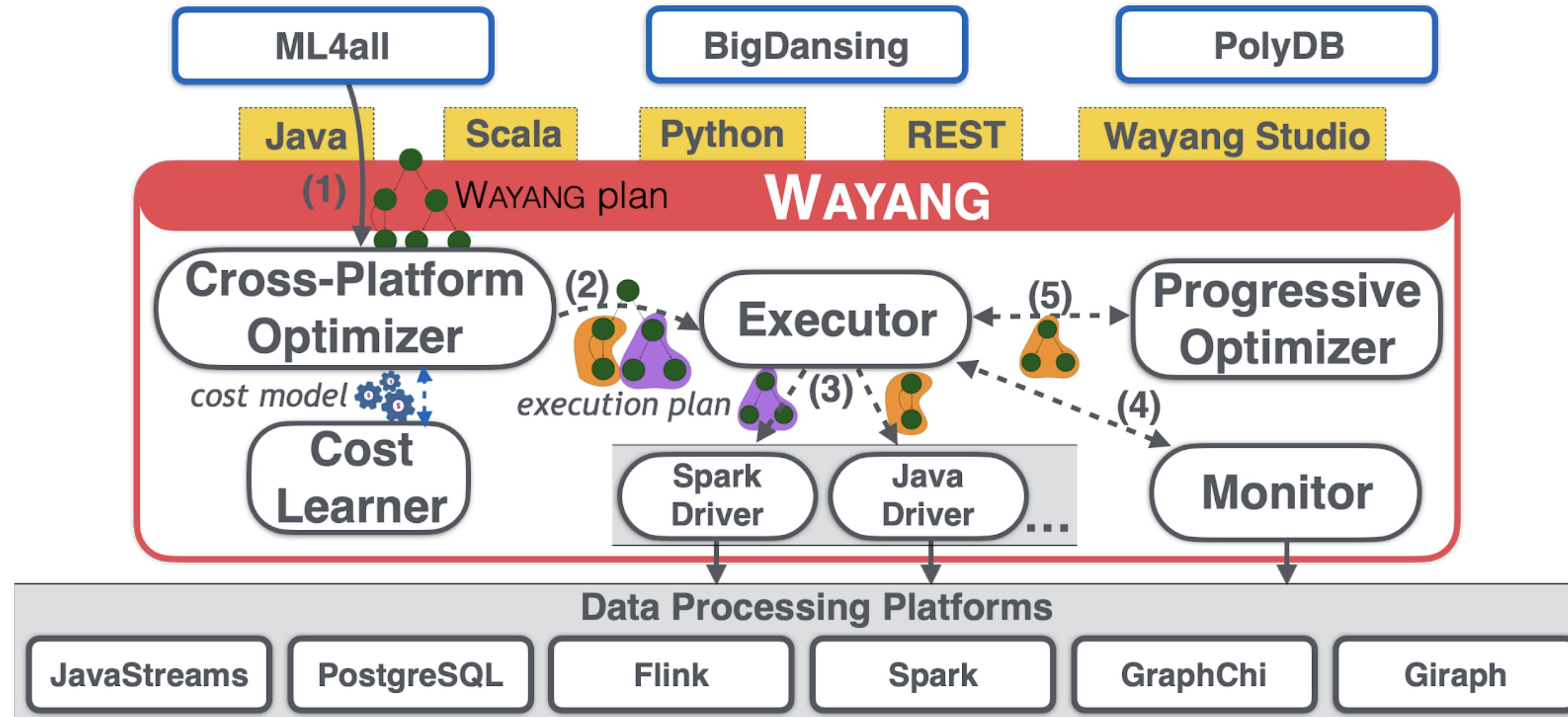
Goal of Wayang

■ RHEEM operator ■ Spark execution operator ■ JavaStreams execution operator
⟩ Input/Output ■ UDF → Data flow ⤵ Broadcast data flow



<https://wayang.apache.org/>

Wayang's Architecture

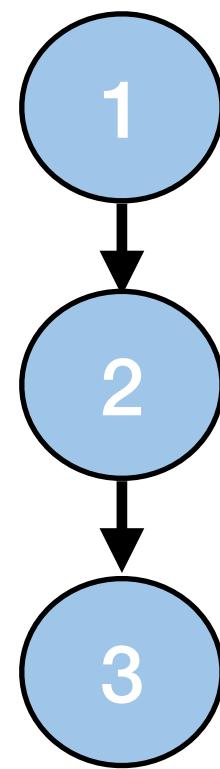


<https://wayang.apache.org/>

Wayang mappings

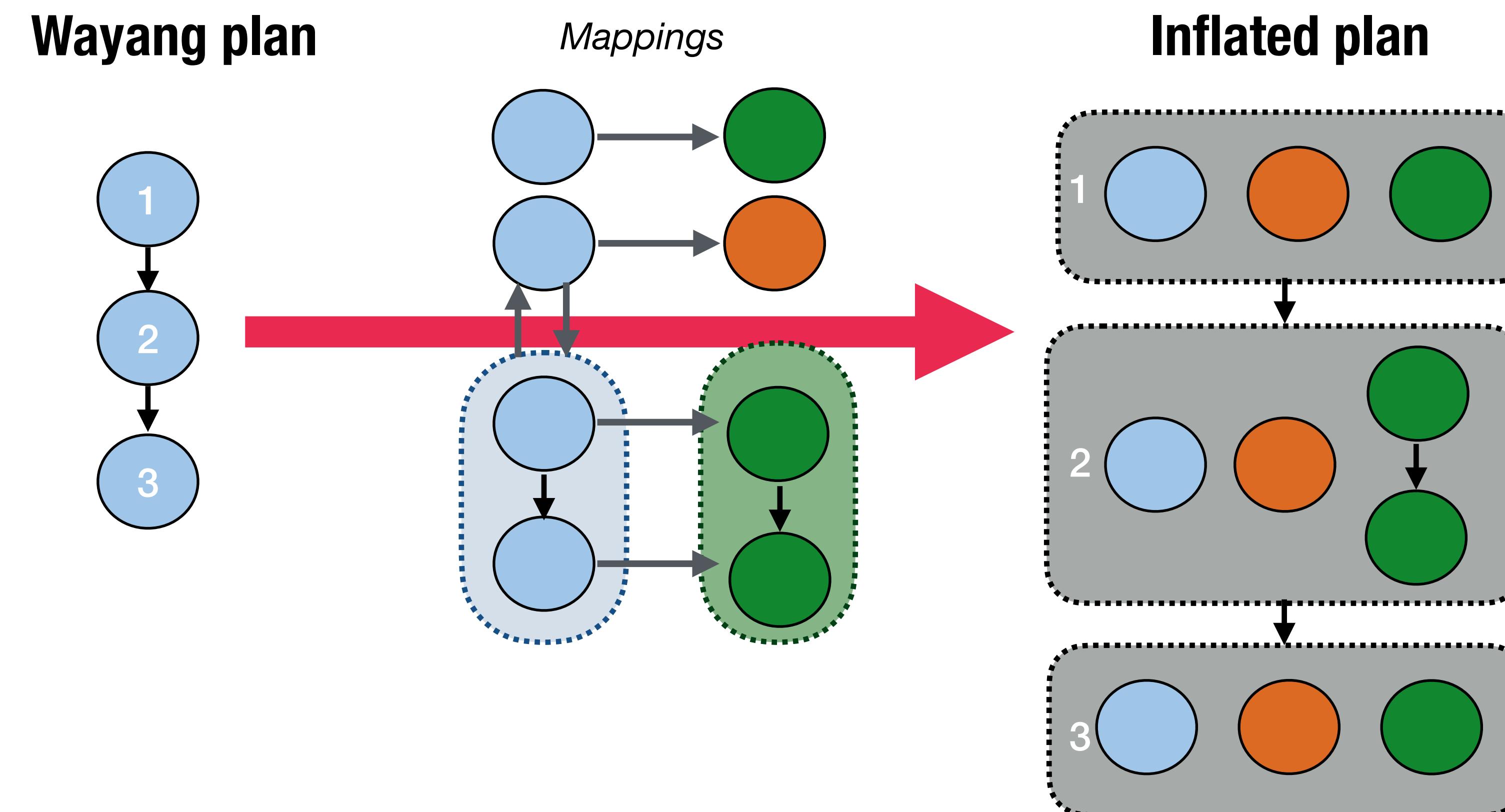
Plan Inflation

Wayang plan



Wayang mappings

Plan Inflation



Decoupling

Applications/
Frontends

Hive

Crunch

MLlib

Mahout

Pig

Cross-Platform System

Processing
Platforms

Hadoop
MR

DBMS

Storm

Spark

Flink

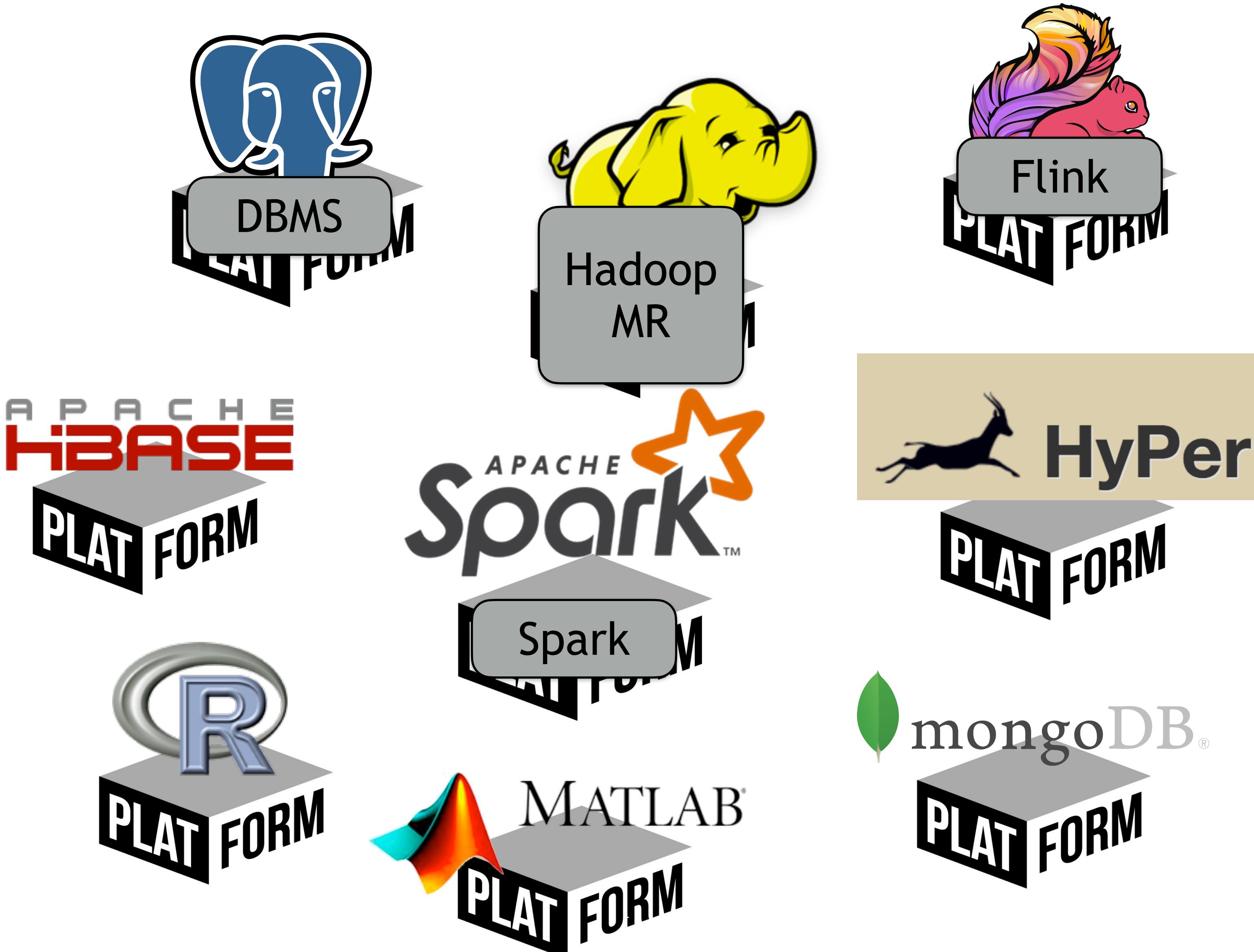
Storage
Engines

HDFS

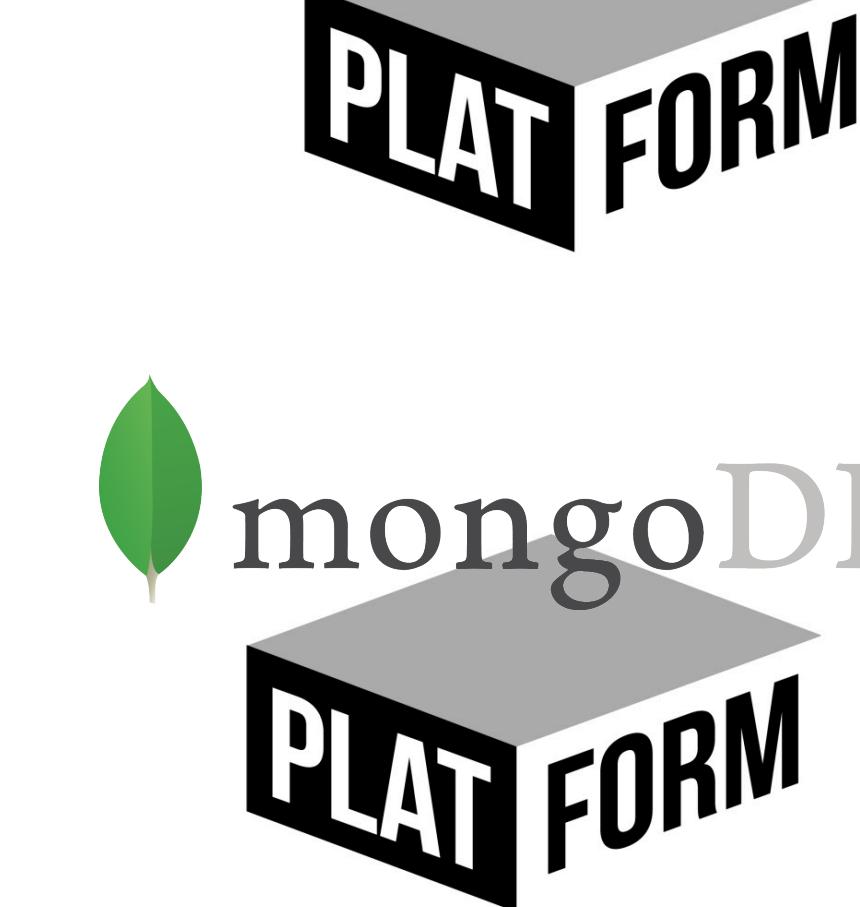
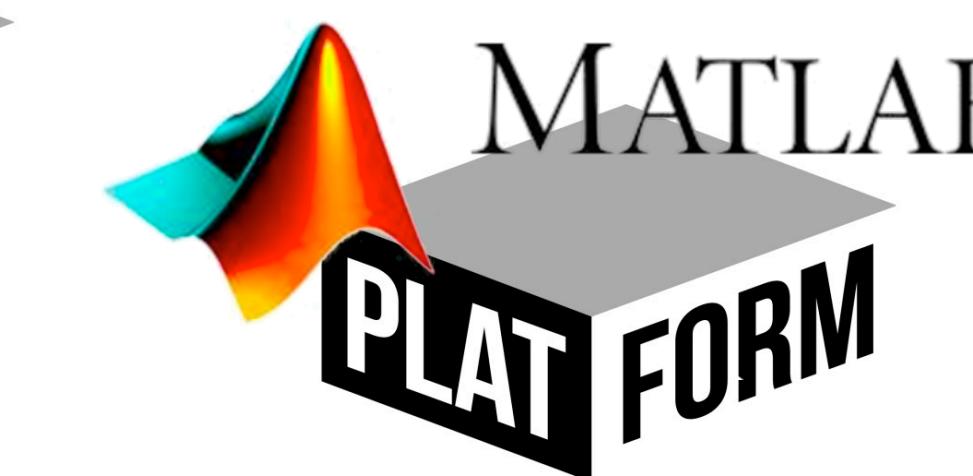
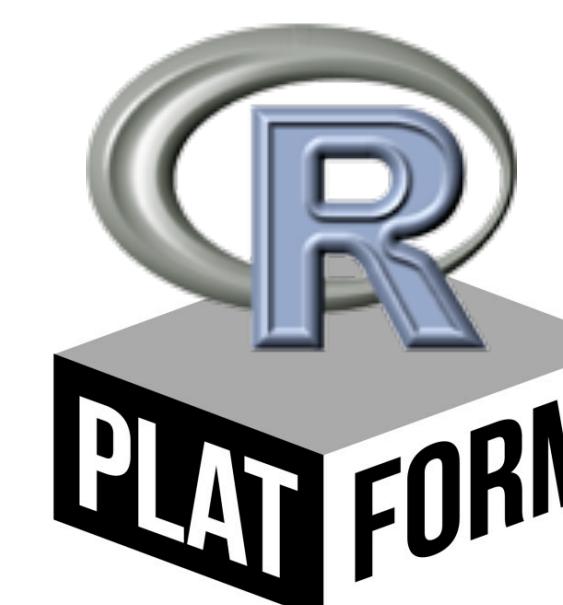
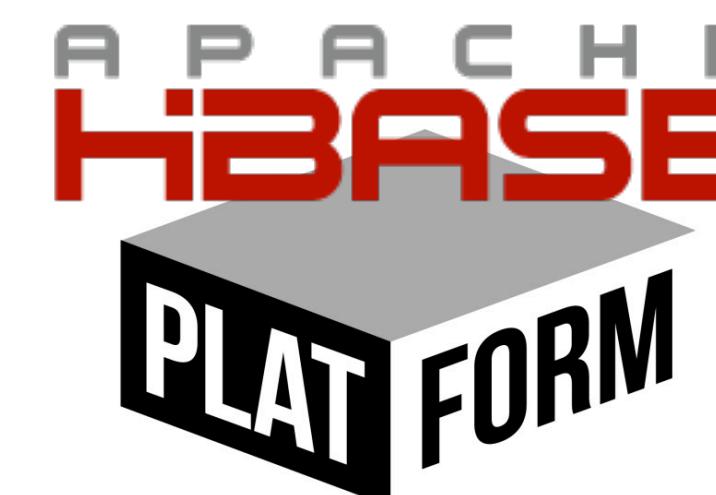
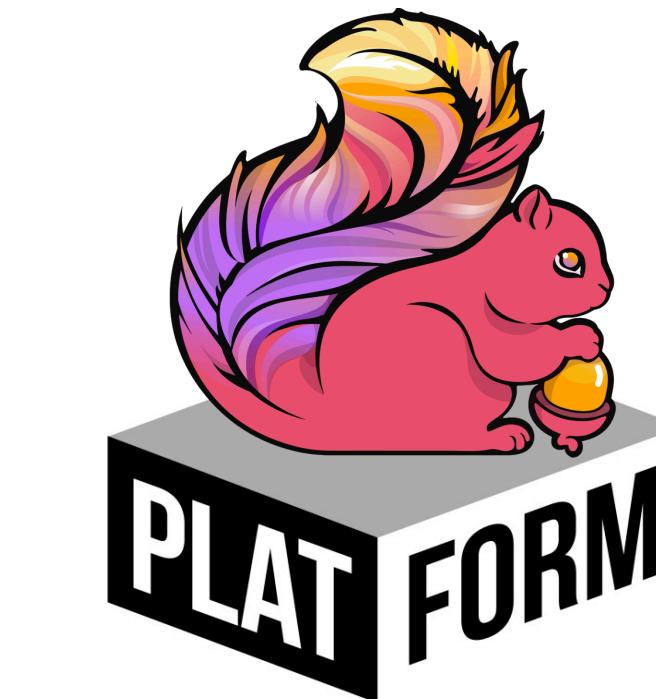
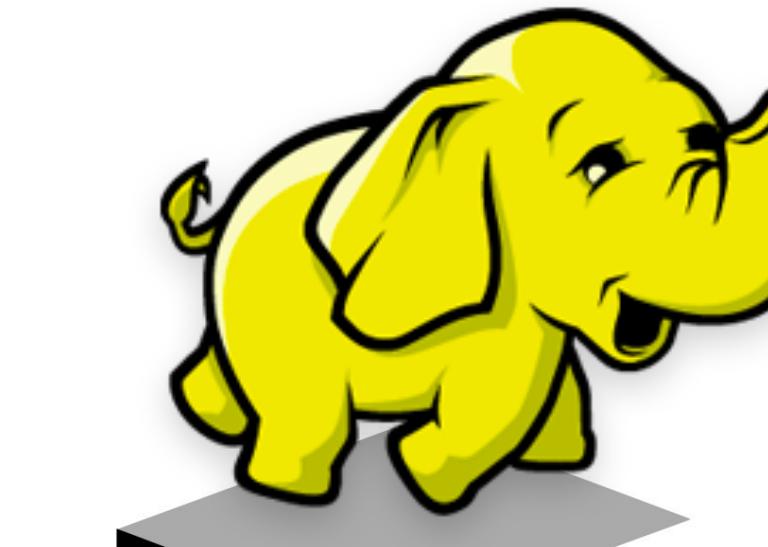
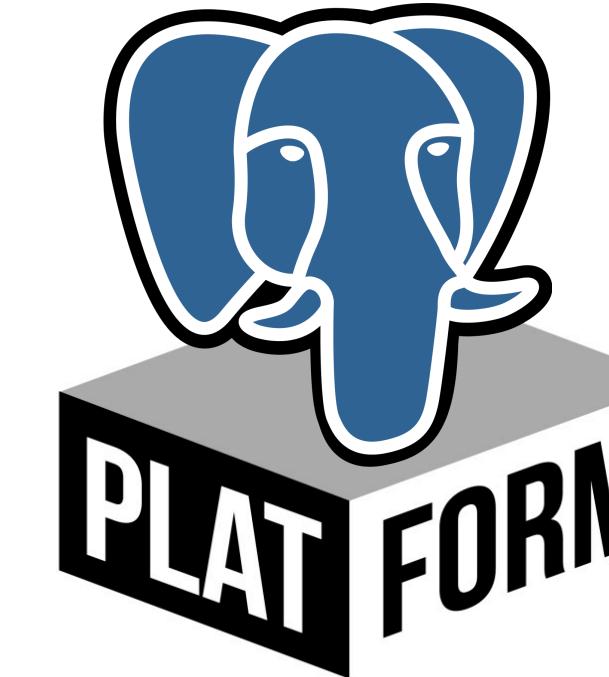
S3

Local FS

Which platform to use?



Which platform to use?



Types of optimization

Types of optimization

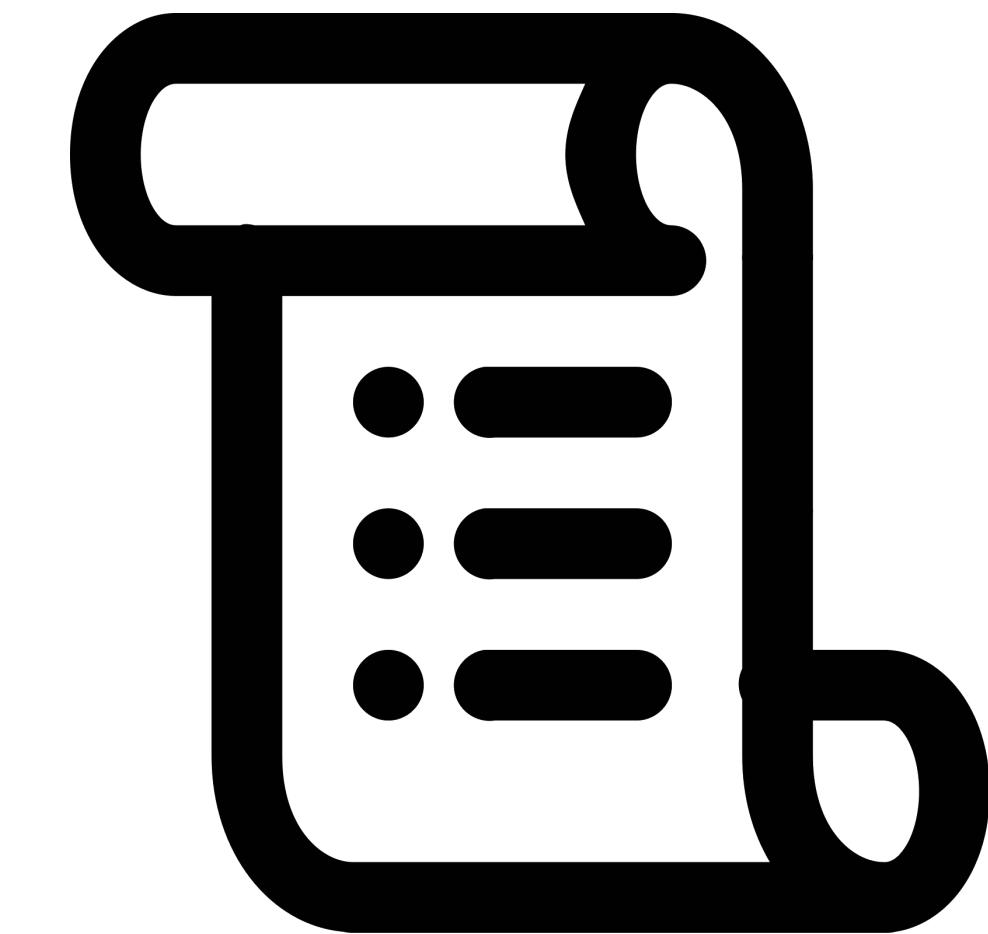


Cost-based

Types of optimization



Cost-based



Rule-based

Types of optimization



Cost-based

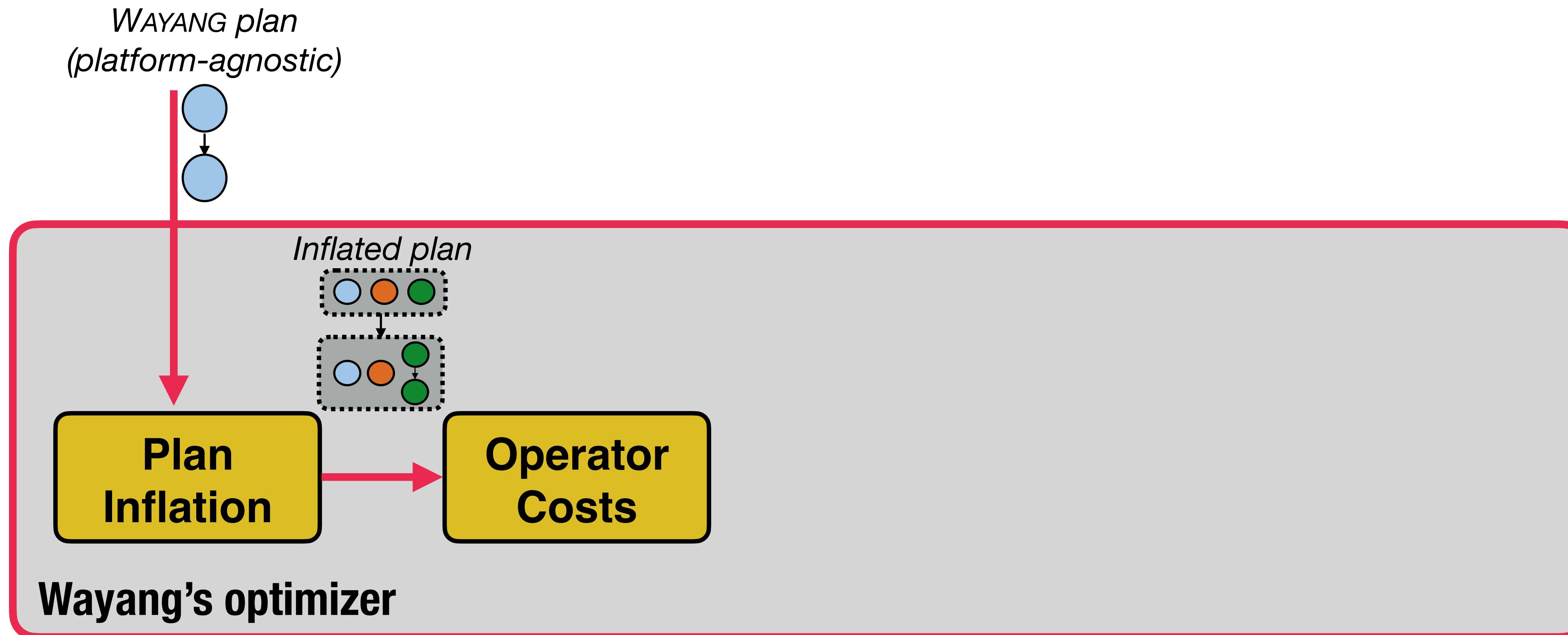


Rule-based

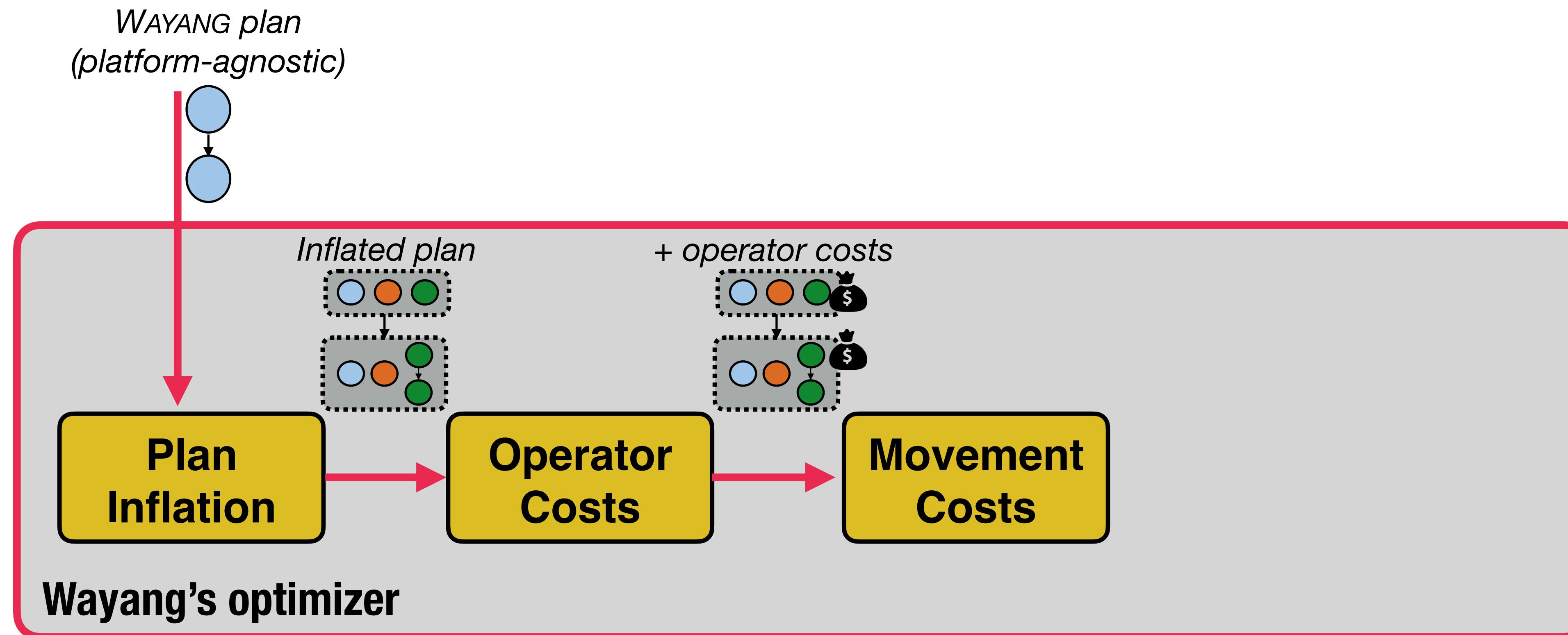
Wayang's Optimizer



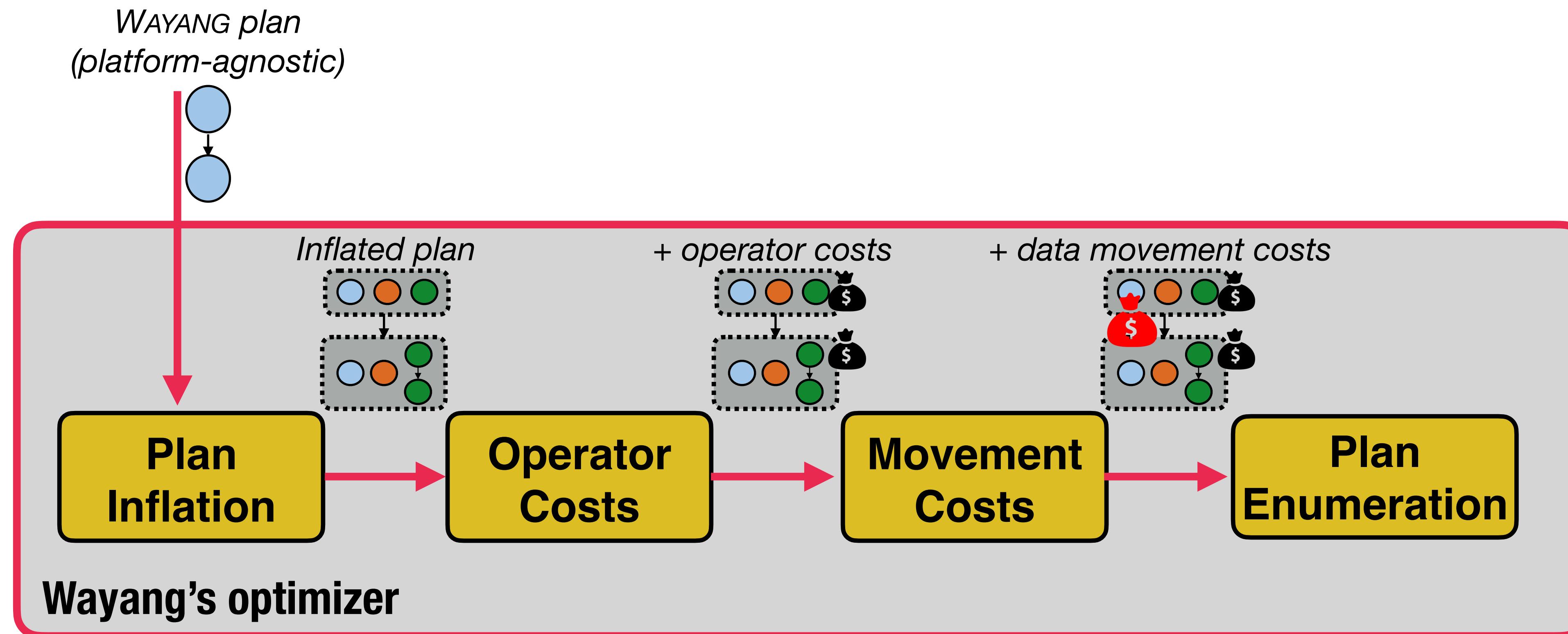
Wayang's Optimizer



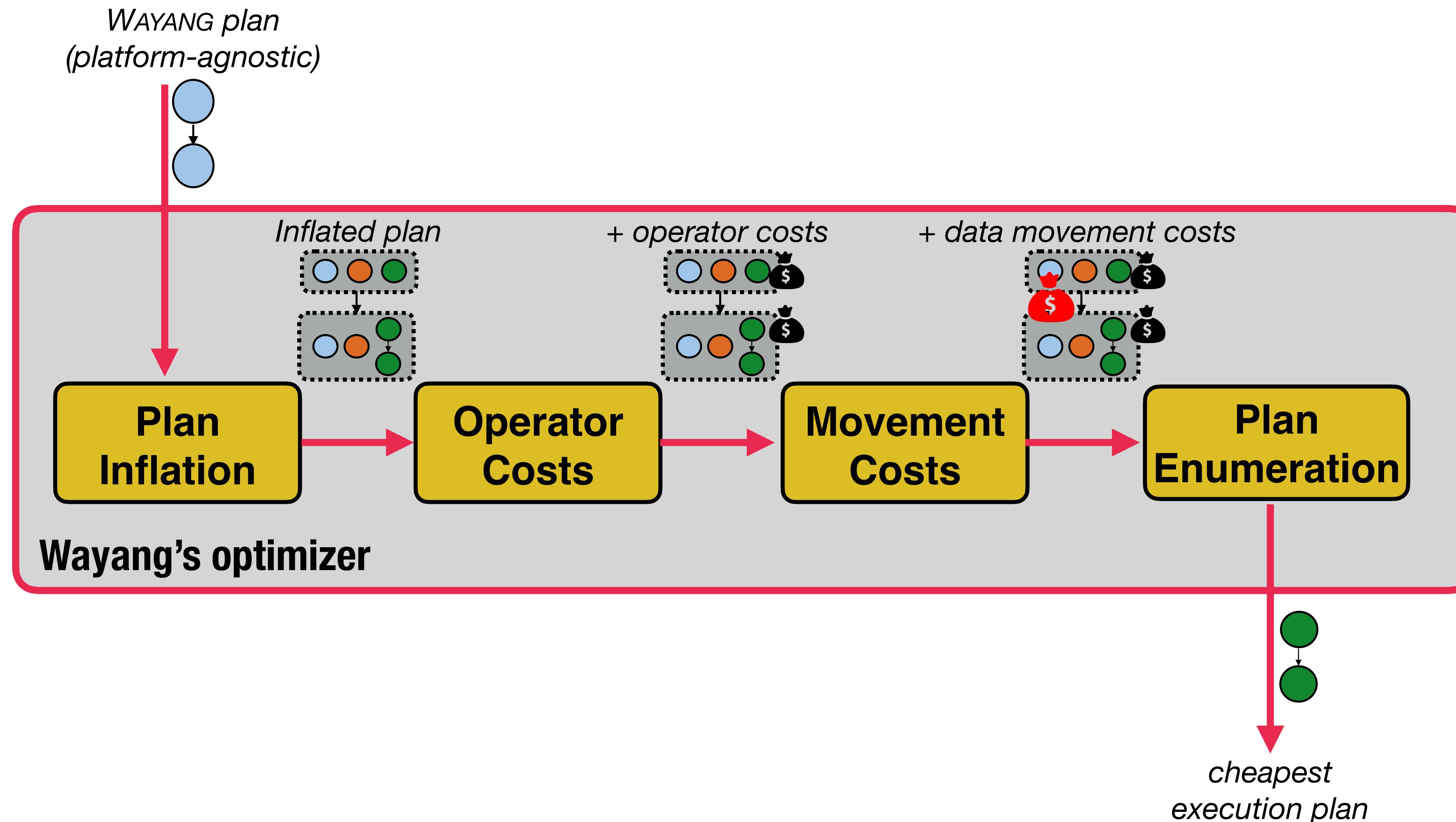
Wayang's Optimizer



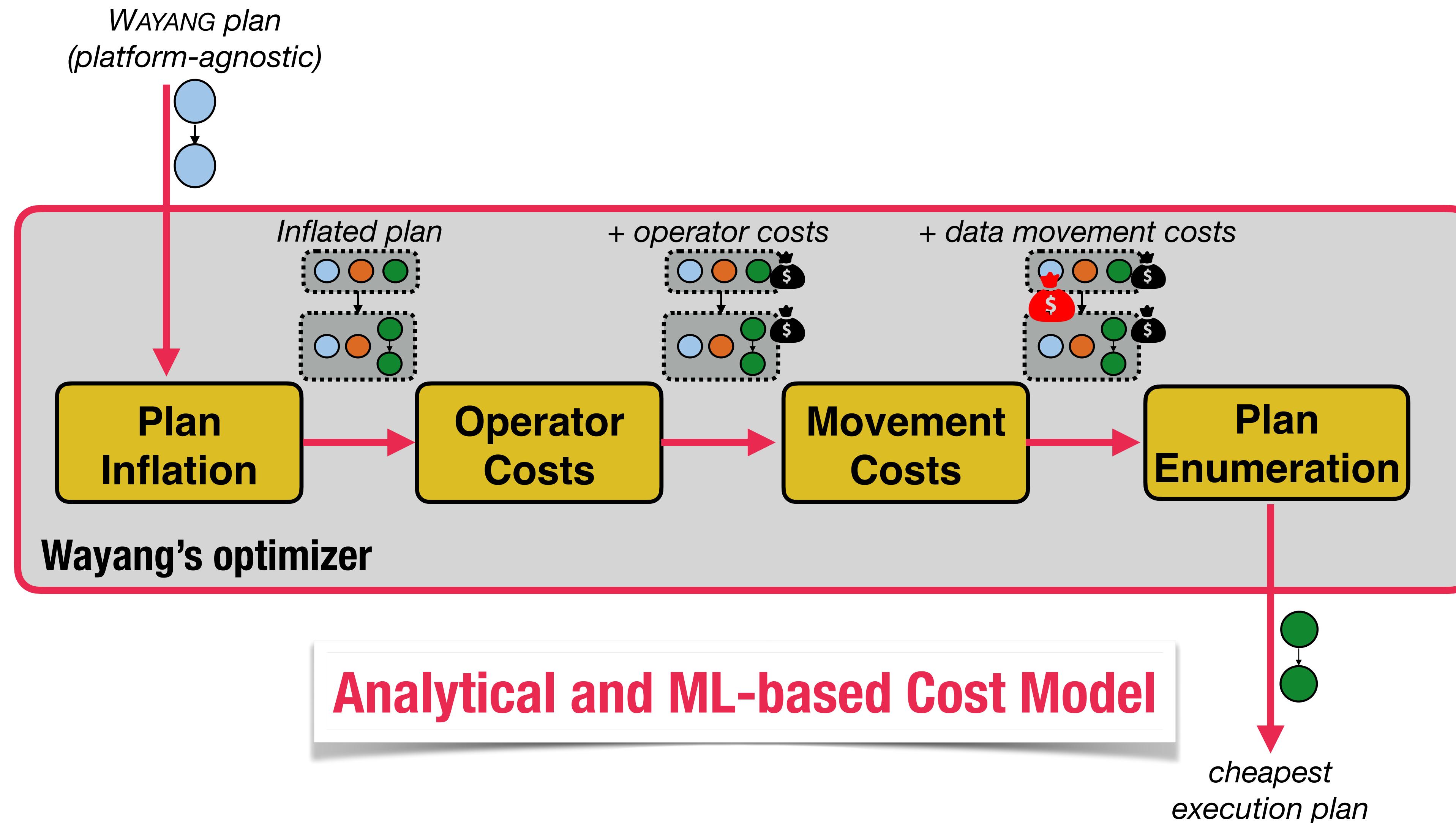
Wayang's Optimizer



Wayang's Optimizer

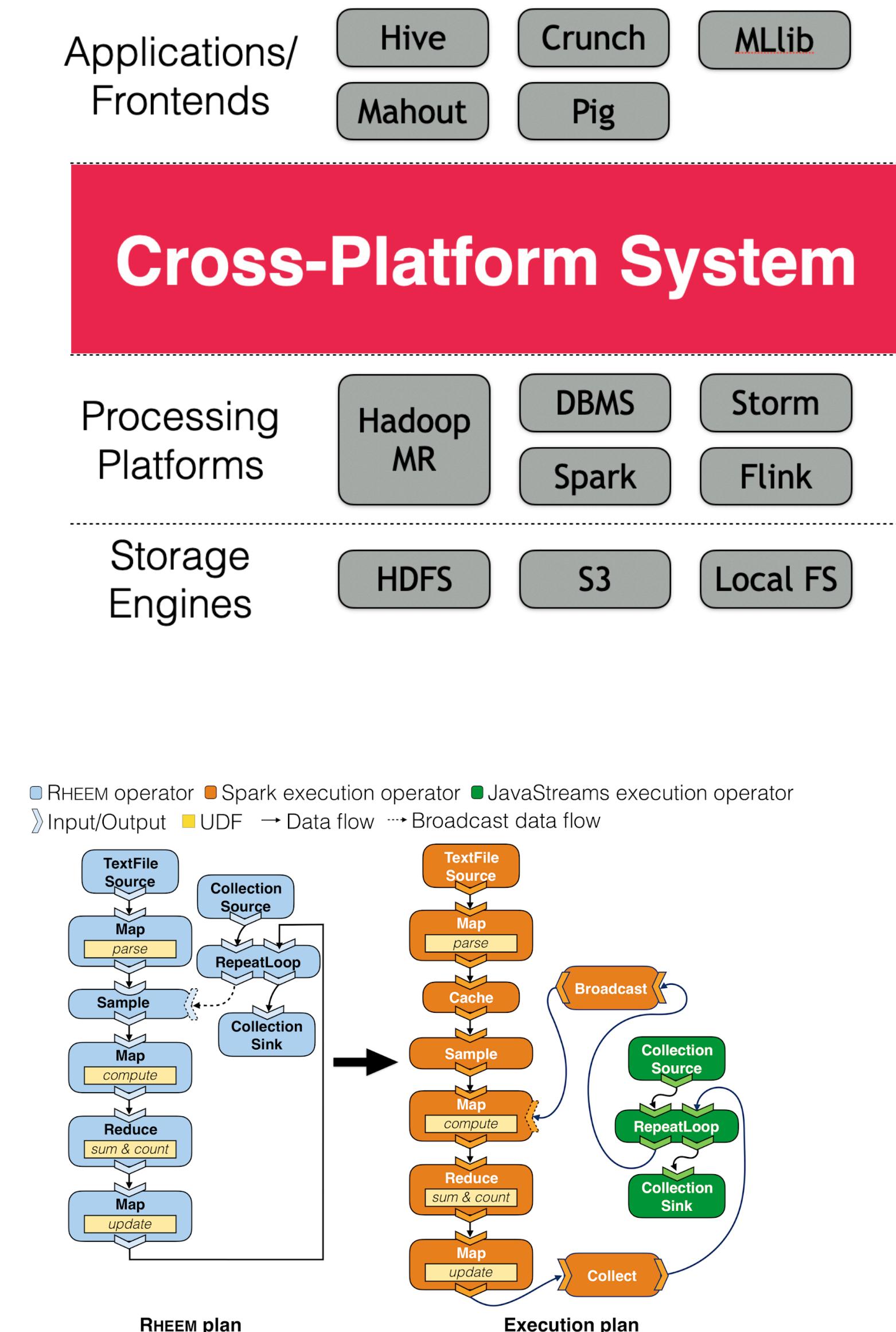


Wayang's Optimizer



Summary

- ◆ Decoupling applications from platforms
- ◆ Automatically determine best execution plan (where each operator has to be executed)
- ◆ Apache Wayang: first open-source cross-platform system



Open projects/thesis on cross-platform data processing

Supervisor: Jorge Quiané

PROPOSAL

Apache Wayang (Big Data)

Do you like open-source systems? Would you like to experience working with an open-source system? Do you want to learn about big data research in practice? Then, this project is for you! We have a number of thesis/project topics under the umbrella of Apache Wayang. Wayang is the first cross-platform framework that allows users to specify their task/query in a system-agnostic manner and Wayang will ...

Supervisors: Jorge Quiané

Semester: Fall 2023

Tags: big data, database, cross-platform data processing, open source, Apache



Data-Intensive Systems and Applications

Readings

- ♦ Spark papers:

- ♦ Matei Zaharia et al.: Spark: Cluster Computing with Working Sets. HotCloud 2010 [\[pdf\]](#)
- ♦ Matei Zaharia et al.: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI 2012: 15-28 [\[pdf\]](#)

- ♦ Wayang papers:

- ♦ RHEEM: Enabling Cross-Platform Data Processing - May The Big Data Be With You! -. Proc. VLDB Endow. 11(11): 1414-1427 (2018) [\[pdf\]](#)