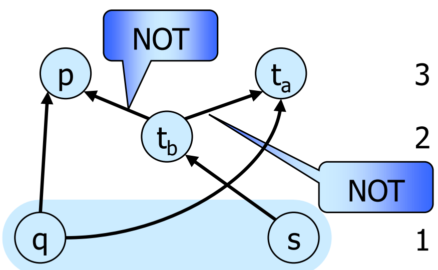
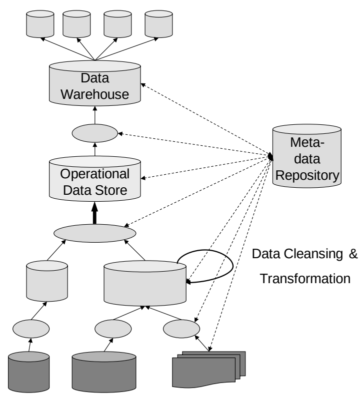
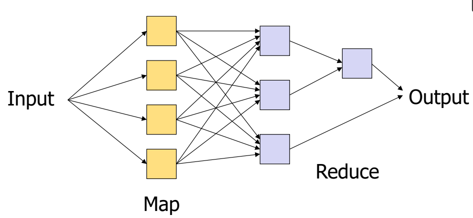
Chapter 5

Advanced Query Processing





?-
:-



Overview

5.1 Query Processing in Big Data Systems

5.2 Query Processing in Deductive Database Systems

5.3 Queries in Data Integration Systems

Literature

Karau H. & Warren, R.: High Performance Spark. O'Reilly, 2017.
Ré, C., & Salihoglu, S.: Topics in DBMS. Course at Stanford University, https://web.stanford.edu/class/cs345d-01/
Bauer, A. & Günzel, H. (ed.) Data-Warehouse-Systeme: Architektur, Entwicklung, Anwendung dpunkt-Verlag, 2001
Ceri, S.; Gottlob, G. & Tanca, L. Logic programming and databases Springer Publishing Company, Incorporated, 2012
Quix, C. Metadata Management for quality-oriented Information Logistics in Data Warehouse Systems (in German) RWTH Aachen University, 2003

5.1 Query Processing in Big Data Systems

- Huge amount of data to be processed
  - For example: Google in 2003
    - 20 billion web pages ➔ 400 TB of data
    - Requires ~4 months to read data (using a disk with ~30 MB/sec)
    - Requires ~1000 disks
    - This for just storing and reading the data, but you want to process the data, e.g.,
      - Process crawled documents
      - Process web request logs
      - Compute page rank
      - Build inverted indexes
      - …

➔ Cannot be done on a single machine, needs massive parallel distributed system

Requirements for a Distributed System

1. Scalable
2. Fault-Tolerant
3. Easy To Program
4. Applicable To Many Problems

Challenges in Distributed Programming

- Lots of programming work
  - communication and coordination
  - work partitioning
  - status reporting
  - optimization
  - locality
- Needs to be repeated for every problem you want to solve
- Needs to be fault-tolerant
  - One server may stay up three years (1,000 days)
  - If you have 10,000 servers, expect to lose 10 a day

Map-Reduce

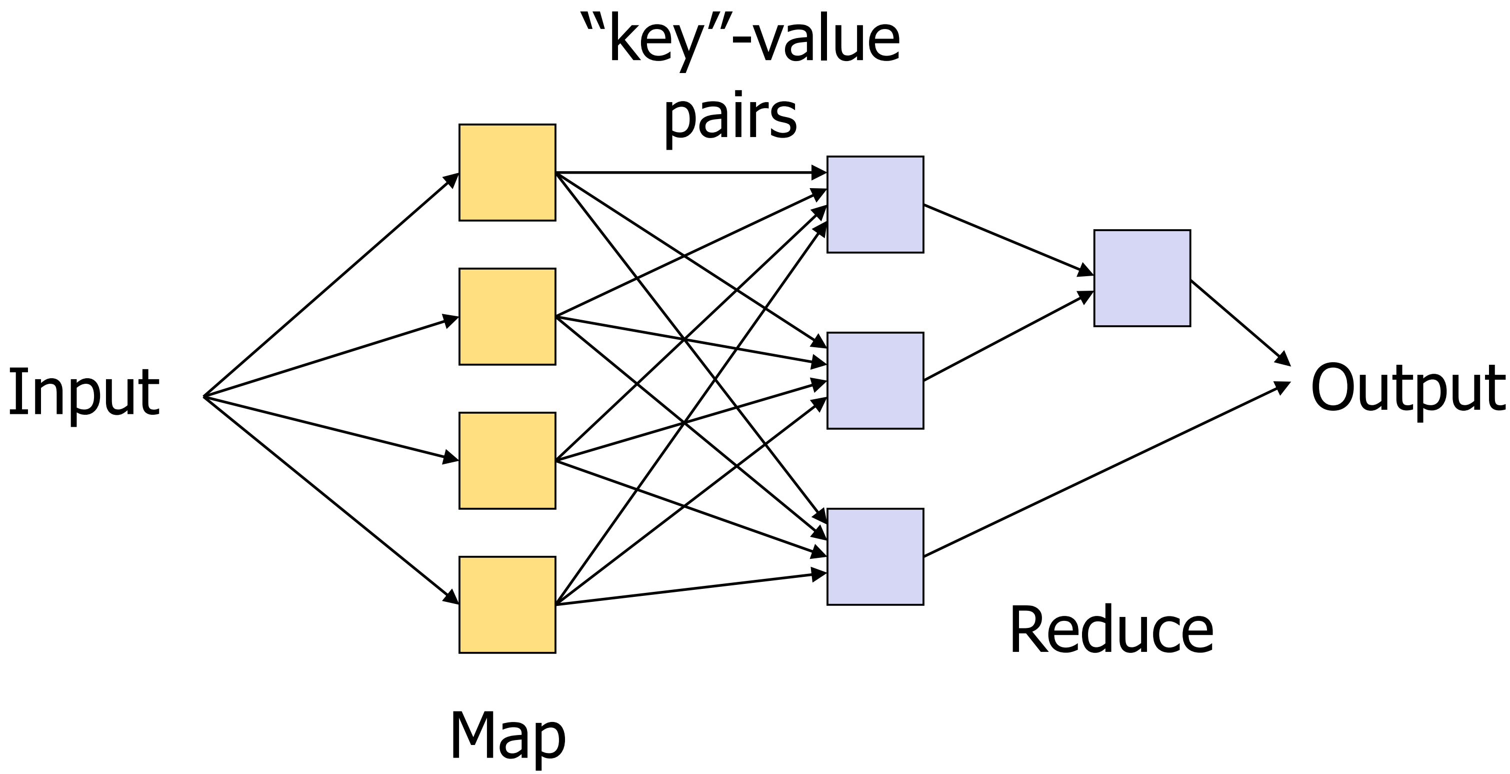
- Programming pattern for parallel computation in distributed system

- Defined by two functions:
  - Map(data) → (key, value)
    - Reads input data and emits key-value pairs
    - Keys are not necessarily unique in emitted pairs

  - Reduce(key, values) → (key, values)
    - Gets input from Map function, a set of values for a single key
    - Input and output structure should be the same
      (some implementations require that output is a single value)
    - Reduce can be called multiple times for the same key

- Map and Reduce jobs can run on different nodes

Map-Reduce



Simple Example for Map-Reduce

(in MongoDB syntax)

Query: Sum of items in customers�much shopping carts, grouped by product ID

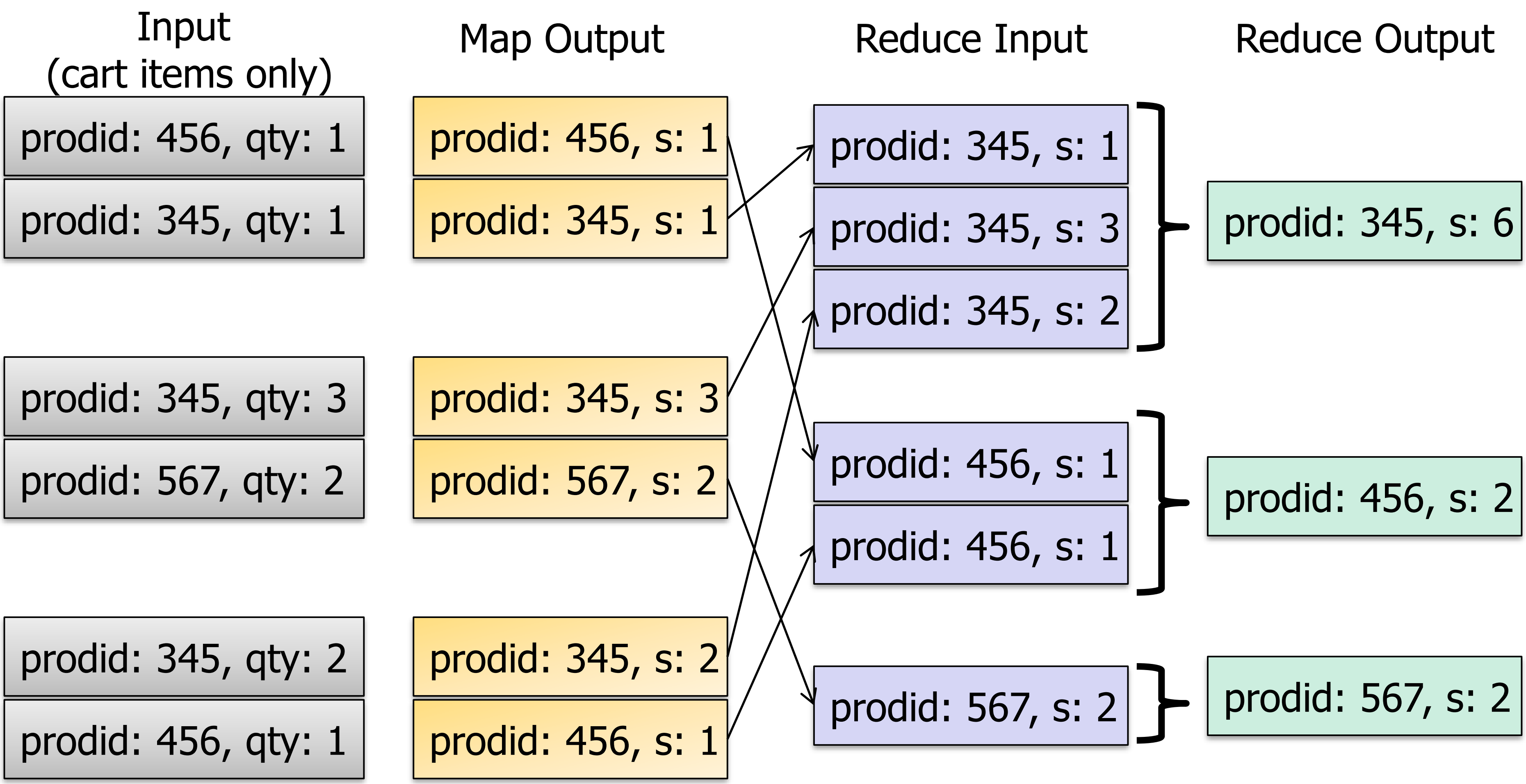
Input: Customer collection

```
{
  firstname: "John",
  lastname: "Doe",
  address: { … },
  cart: [  { prodid: 3456,
              qty: 2},
           { prodid: 6789,
              qty: 1} ]
}
```

```
map=function() {
  if(this.cart!=null) {
    this.cart.forEach(function(item) {
      emit(item.prodid, { sum: item.qty });
    });
  }
}
```

```
reduce=function(key, values) {
  var s=0;
  values.forEach(function(value) {
    s+=value.sum;
  });
  return ( { sum: s} );
}
```

Example


Input
(cart items only)
Map Output
Reduce Input
Reduce Output
prodid: 456, qty: 1
prodid: 345, qty: 1
prodid: 345, qty: 3
prodid: 567, qty: 2
prodid: 345, qty: 2
prodid: 456, qty: 1
prodid: 456, s: 1
prodid: 345, s: 1
prodid: 345, s: 3
prodid: 567, s: 2
prodid: 345, s: 2
prodid: 456, s: 1
prodid: 345, s: 1
prodid: 345, s: 3
prodid: 345, s: 2
prodid: 456, s: 1
prodid: 456, s: 1
prodid: 567, s: 2
prodid: 345, s: 6
prodid: 456, s: 2
prodid: 567, s: 2

A More Complex Example (1)
Join between customers and products

Input: Customers + products collection

```
{
  firstname: "John",
  lastname: "Doe",
  address: { … },
  cart: [ { prodid: 3456,
            qty: 2},
          { prodid: 6789,
            qty: 1} ]
}
{
  prodid: 3456,
  price: 34
}
{
  prodid: 6789,
  price: 23
}
```

Query: The sum of items and their value in all shopping carts, grouped by product ID

```
map=function() {
  if(this.cart!=null) {
    this.cart.forEach(function(item) {
      emit(item.prodid, { qty: item.qty,
                          price: null,
                          total: null });
    });
  }
  if(this.prodid!=null) {
    emit(this.prodid, {qty: 0,
                       price: this.price,
                       total: 0 });
  }
}
```

A More Complex Example (2)

Join between customers and products

Input: Customers + products collection

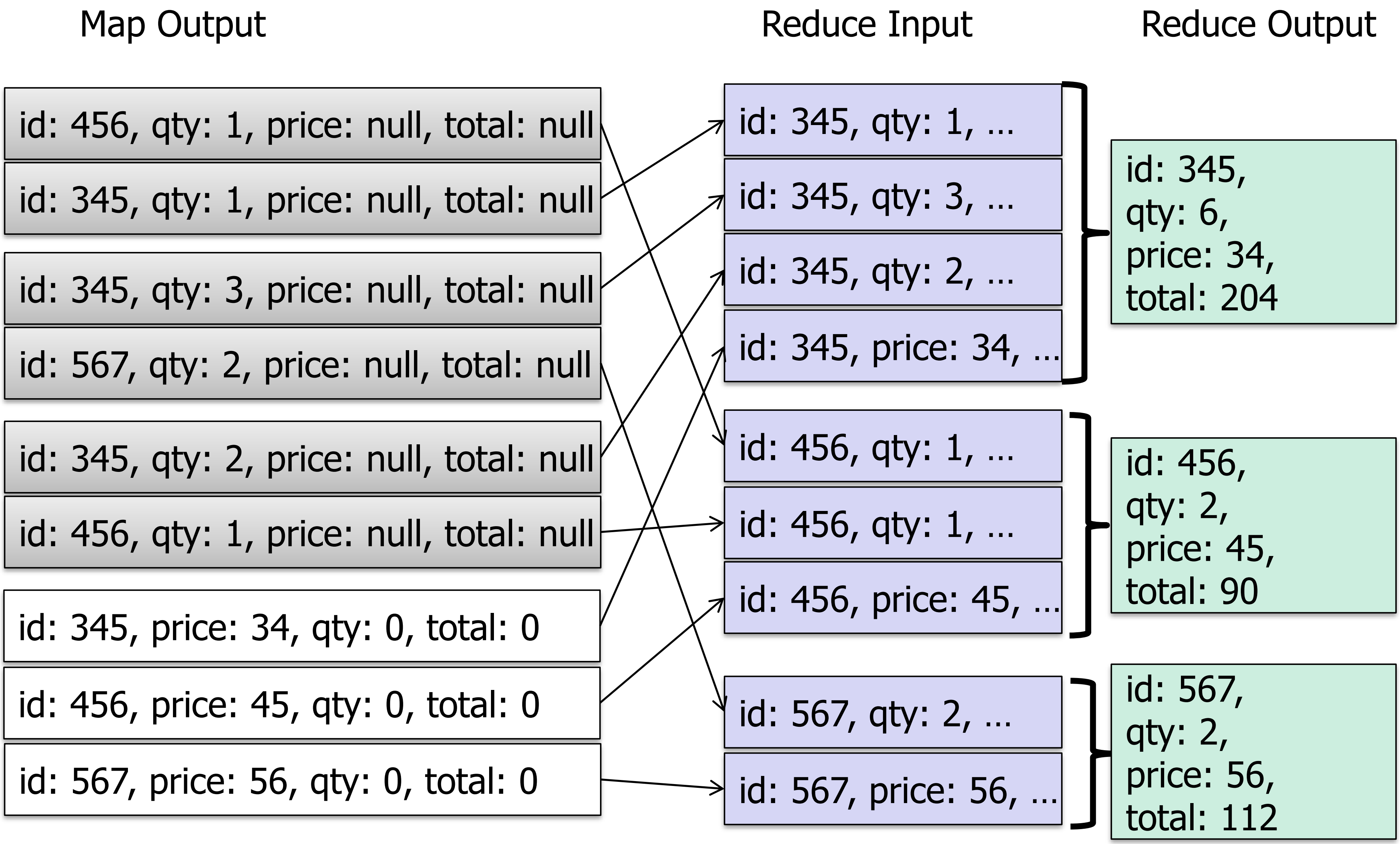
```
{
  firstname: "John",
  lastname: "Doe",
  address: { … },
  cart: [  { prodid: 3456,
               qty: 2},
             { prodid: 6789,
               qty: 1} ]
}
{
  prodid: 3456,
  price: 34
}
{
  prodid: 6789,
  price: 23
}
```

Query: The sum of items and their value in all shopping carts, grouped by product ID

```
map=function() {
  i
  }
  }
  i
  }
}
```

```
reduce=function(key, values) {
  var res = { qty : 0, price: null, total: null };
  values.forEach(function(value) {
    res.qty+=value.qty;

    if(res.price!=null) {
      res.total+=res.price*value.qty;
    }

    if(res.price==null && value.price!=null) {
      res.price=value.price;
      res.total=res.price * res.qty;
    }
  });
  return res;
}
```

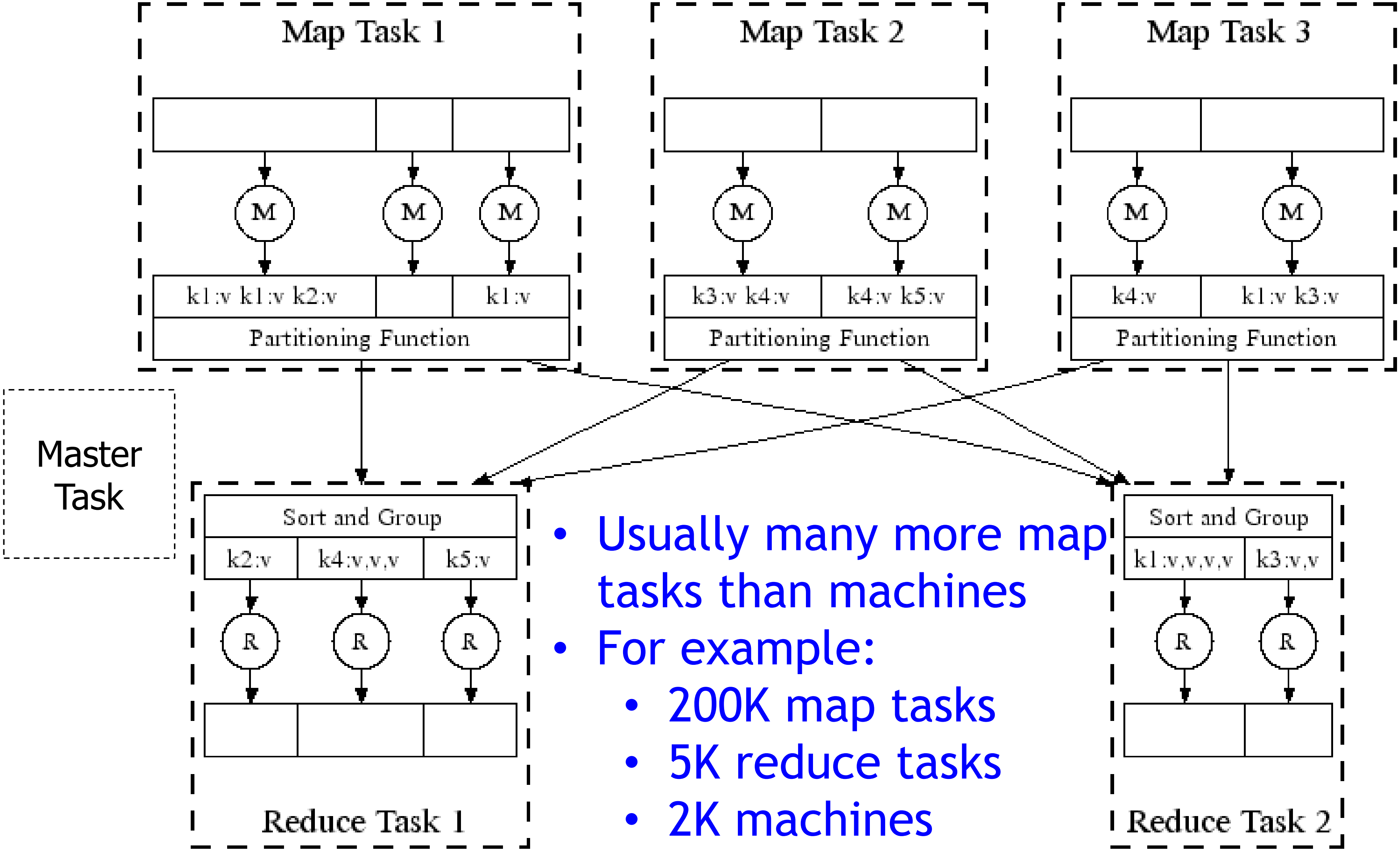
A More Complex Example (3)
Join between customers and products



Key Points of Map-Reduce

- Map-Reduce: pattern enabling parallel query processing over a cluster
- Many implementations, e.g., in Hadoop with distributed file system or NoSQL systems with database query processing
- No coordination is required between individual map and reduce jobs
- Sharding & replication fits nicely into Map-Reduce pattern
  - Parallelism is increased: Map tasks can be distributed to different shards or their replicas
  - Availability is increased: if a node is not available, replica can take over the job
- Mapping of input data is important
  - Key determines grouping
  - Data structure of value is output

MapReduce Execution



- Usually many more map tasks than machines
- For example:
  - 200K map tasks
  - 5K reduce tasks
  - 2K machines

Fault-Tolerance: Handled via re-execution

- On worker failure
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress map tasks
  - Re-execute in progress reduce tasks
  - Task completion committed through master
- Master failure
  - Is much more rare
  - Fail-over to secondary master nodes (e.g., name node, resource manager)

Map Reduce Limitations

- Many queries/computations need multiple MR jobs
- 2-stage computation too rigid
- Example: Find the top 10 most visited pages in each category

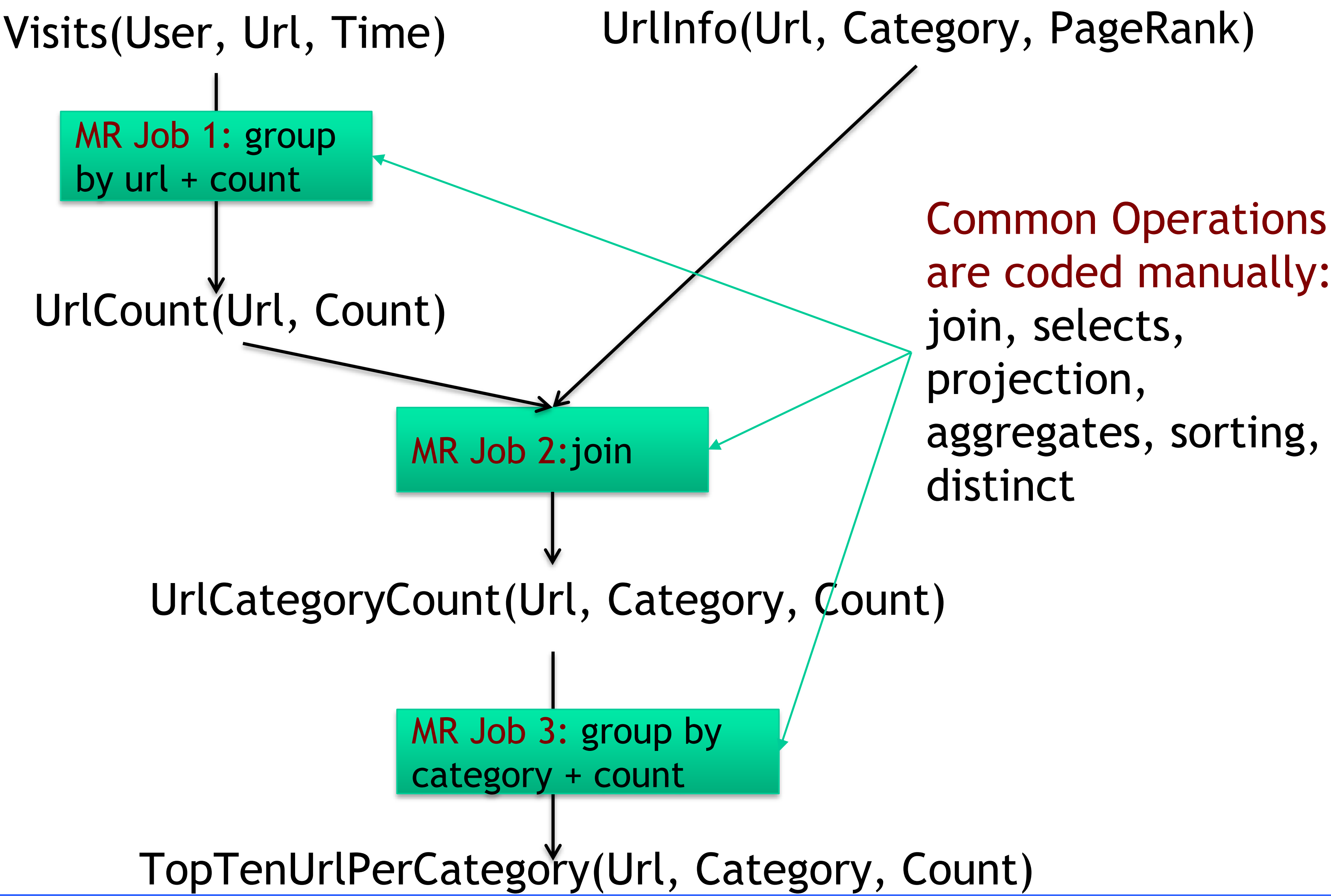
Visits

User	Url	Time
Amy	cnn.com	8:00
Amy	bbc.com	10:00
Amy	flickr.com	10:05
Fred	cnn.com	12:00

UrlInfo

Url	Category	PageRank
cnn.com	News	0.9
bbc.com	News	0.8
flickr.com	Photos	0.7
espn.com	Sports	0.9

Example: Top 10 most visited pages in each category



Map Reduce is not a good model for data processing

- Required
  - Support for algebra operations: join, selection, projection, group by, …
  - More abstract language to implement data transformations and queries

➔ High-level languages that are compiled into Map-Reduce jobs

  ➔ Apache Pig (Pig Latin)

  ➔ Apache Hive (HiveQL)

Apache Pig

- High-level language for analyzing large data sets
- Apache project since 2007, mainly supported by Twitter and Hortonworks
- Pig Latin: Procedural language with algebra-like operations (join, selection, projection, …)
- Workflows can be defined as step-by-step procedural scripts
- Pig Latin can be compiled to jobs on
  - Hadoop
  - Tez
  - Spark

Based on: Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins: Pig latin: a not-so-foreign language for data processing. SIGMOD Conference 2008: 1099-1110

Pig Latin Example

```
visits          = load '/data/visits' as (user, url, time);
gVisits         = group visits by url;
urlCounts  = foreach gVisits generate url, count(visits);

urlInfo         = load '/data/urlInfo' as (url, category, pRank);
urlCategoryCount = join urlCounts by url, urlInfo by url;

gCategories = group urlCategoryCount by category;
topUrls = foreach gCategories generate top(urlCounts,10);

store topUrls into '/data/topUrls';
```

Pig Latin Example

```
visits           = load '/data/visits' as (user, url, time);
gVisits          = group visits by url;
urlCounts = foreach gVisits generate url, count(visits);

urlInfo          = load '/data/urlInfo' as (url, category, pRank);
urlCategoryCount = join urlCounts by url, urlInfo by url;

gCategories = ...gory;
topUrls = fore...Counts,10);

store topUrls into '/data/topUrls';
```

Operates directly over files

Pig Latin Example

```
visits           = load '/data/visits' as (user, url, time);
gVisits          = group visits by url;
urlCounts = foreach gVisits generate url, count(visits);

urlInfo          = load '/data/urlInfo' as (url, category, pRank);
urlCategoryCount = join urlCounts by url, urlInfo by url;

gCategories                                            gory;
topUrls = fo                                           Counts,10);

store topUrls into '/data/topUrls';
```

Schemas optional;
Can be assigned dynamically
➔ Schema-on-Read

Pig Latin Example

User-defined functions (UDFs) can be used in every construct

- Load, Store
- Group, Filter, Foreach

```
visits ... ime);
gVisits
urlCount ... (visits);

urlInfo ... ory, pRank);
urlCategoryCount = join urlCounts by url, urlInfo by url;

gCategories = group urlCategoryCount by category;
topUrls = foreach gCategories generate top(urlCounts,10);

store topUrls into '/data/topUrls';
```

Pig Latin Example

```
visits          = load '/data/visits' as (user, url, time);
gVisits         = group visits by url;
urlCounts = foreach gVisits generate url, count(visits);
```

MR Job 1

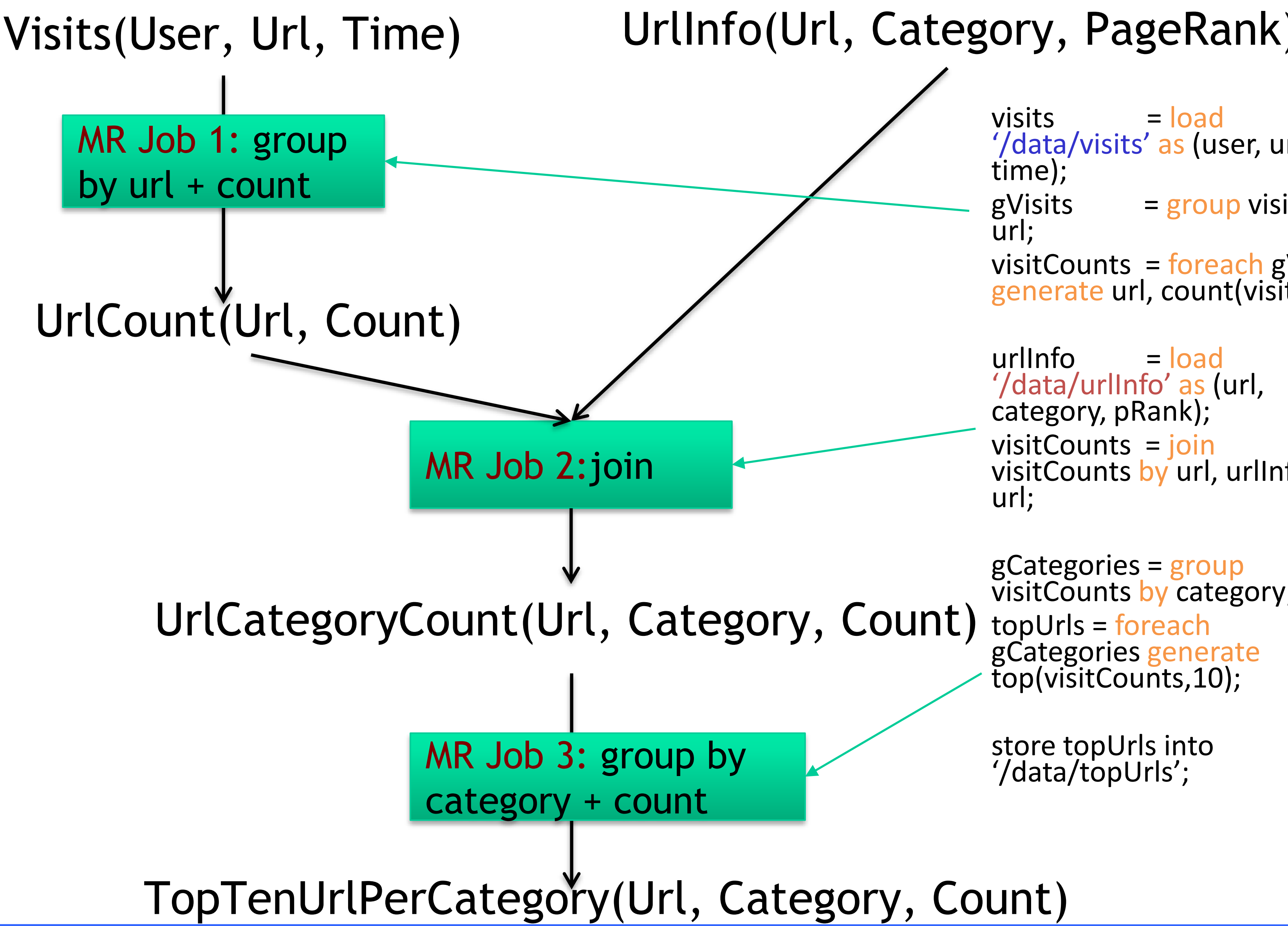
```
urlInfo         = load '/data/urlInfo' as (url, category, pRank);
urlCategoryCount = join urlCounts by url, urlInfo by
```

MR Job 2

```
gCategories = group urlCategoryCount by categor
topUrls = foreach gCategories generate top(urlCounts,10);

store topUrls into '/data/topUrls';
```

MR Job 3

Execution of Pig Latin Example: Compiles also to Map-Reduce jobs




```
visits          = load '/data/visits' as (user, url, time);
gVisits         = group visits by url;
visitCounts  = foreach gVisits generate url, count(visits);

urlInfo         = load '/data/urlInfo' as (url, category, pRank);
visitCounts  = join visitCounts by url, urlInfo by url;

gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts,10);

store topUrls into '/data/topUrls';
```

Apache Hive

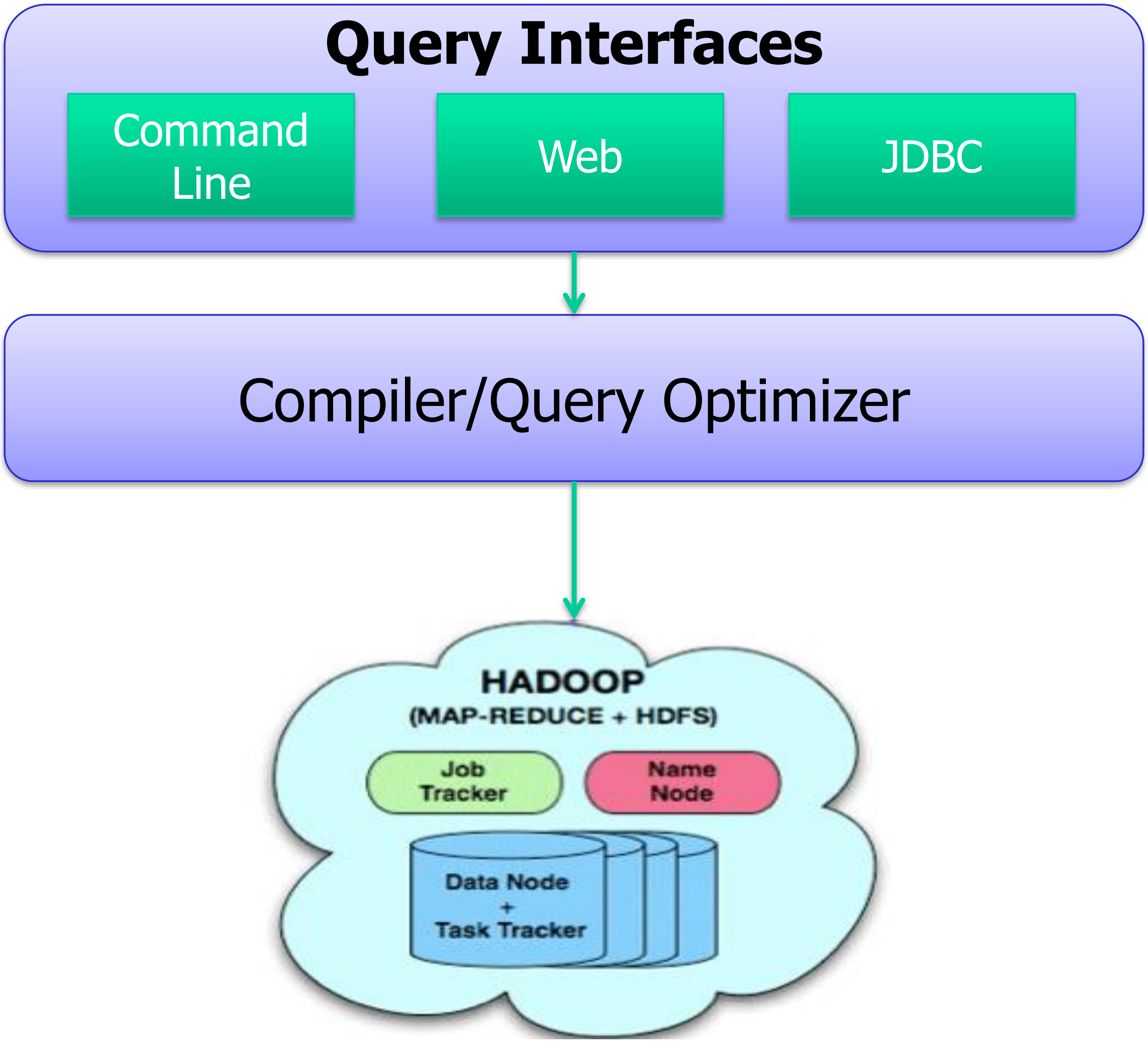
- Data warehouse software for large datasets in distributed storage
- Hive-QL: SQL-like declarative language
  - e.g., SELECT *, INSERT INTO, GROUP BY, SORT BY
- Compiles to
  - Hadoop
  - Tez
  - Spark

Based on: Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, Raghotham Murthy: Hive - A Warehousing Solution Over a Map-Reduce Framework. PVLDB 2(2): 1626-1629 (2009)

Hive Example

```
INSERT TABLE UrlCounts
(SELECT url,  count(*) AS count
 FROM Visits
 GROUP BY url)

INSERT TABLE UrlCategoryCount
(SELECT url, count, category
FROM UrlCounts JOIN UrlInfo ON (UrlCounts.url = UrlInfo
.url))

SELECT category, topTen(*)
FROM UrlCategoryCount
GROUP BY category
```

Apache Hive Architecture



Hive Execution Example: Compiles also to Map-Reduce jobs

Visits(User, Url, Time)

UrlInfo(Url, Category, PageRank)

MR Job 1: group by url + count

```
INSERT TABLE UrlCounts
(SELECT url,  count(*) AS count
 FROM Visits
 GROUP BY url)
```

UrlCount(Url, Count)

MR Job 2:join

```
INSERT TABLE UrlCategoryCount
(SELECT url, count, category
FROM UrlCounts JOIN UrlInfo ON
(UrlCounts.url = UrlInfo .url))
```

UrlCategoryCount(Url, Category, Cou

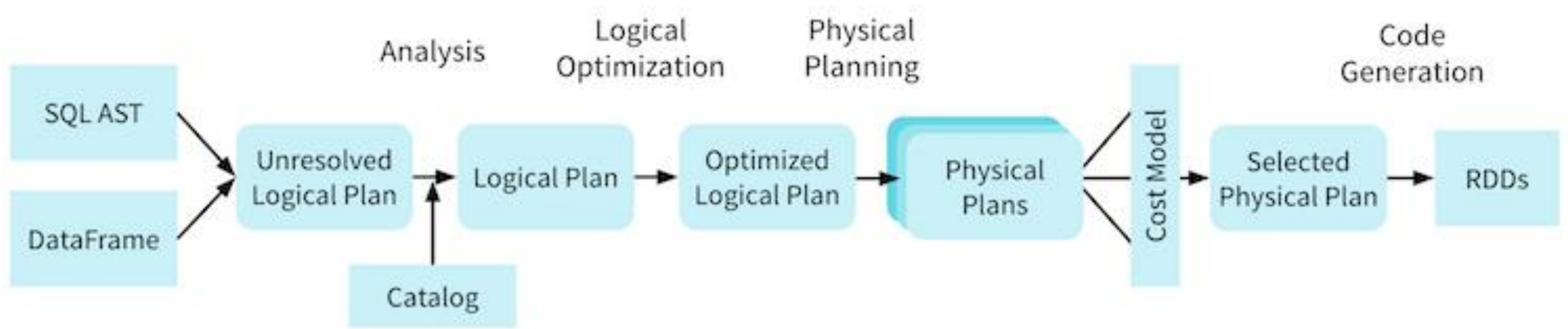
```
SELECT category, topTen(*)
FROM UrlCategoryCount
GROUP BY category
```

MR Job 3: group by category + count

TopTenUrlPerCategory(Url, Category, Count)

Apache Spark

- See also chapter 4.2.2
- Queries can be also specified in Spark SQL (SQL dialect of Apache Spark)
- DataFrames represent relational tables, implemented as RDDs
- Query processing are transformations & actions on RDDs/DataFrames



Apache Spark Example

```
var visits=spark.read.format(„com.databricks.spark.csv“)
        .option(„header“,“true“).load(„visits.csv“)
var urlinfo=spark.read.format(„com.databricks.spark.csv“)
        .option(„header“,“true“).load(„urlinfo.csv“)

visits.createOrReplaceTempView(„visits“)
urlinfo.createOrReplaceTempView(„urlinfo“)

val q=spark.sql(„SELECT v.url, u.category, count(*)
                FROM visits v, urlinfo u
                WHERE v.url=u.url
                GROUP BY v.url, u.category“)

q.explain()
```

Create tables for SQL queries

Filtering of top 10 URLs by category has to be done separately

Show query plan

Apache Spark Query Plan

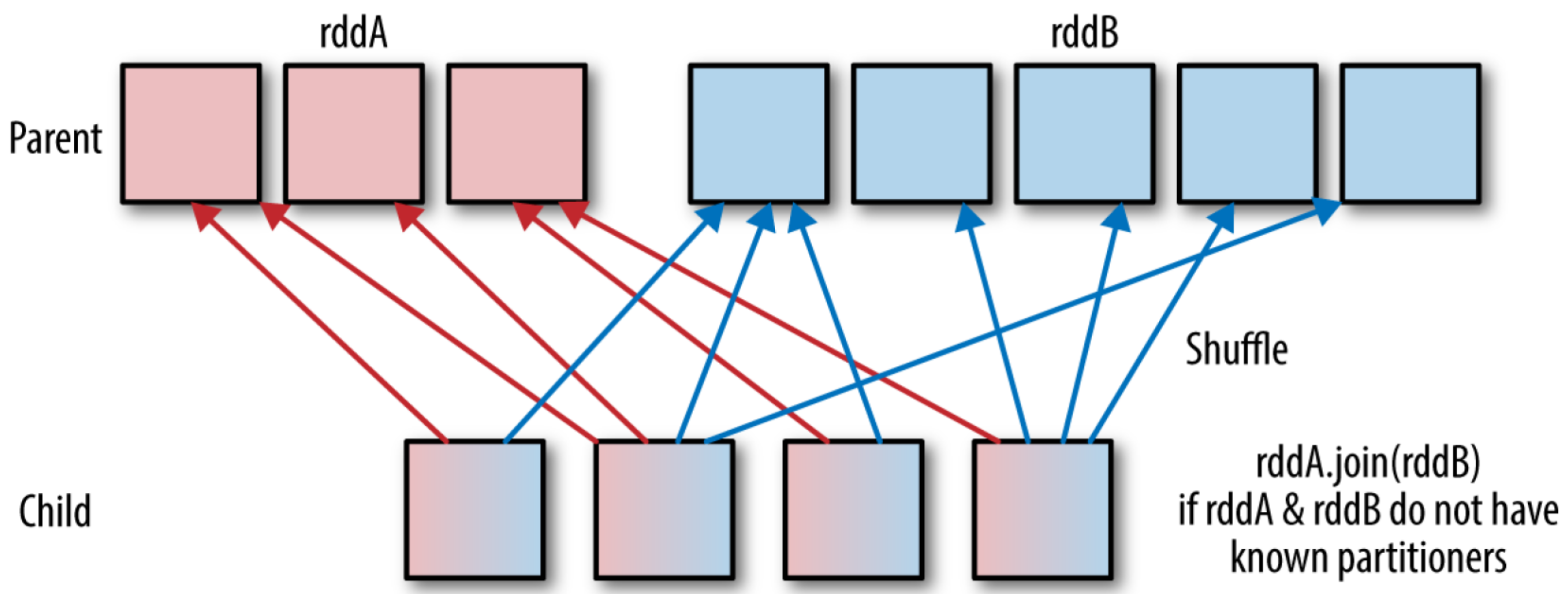
```
scala> q.explain()
== Physical Plan ==
*HashAggregate(keys=[url#13, category#40], functions=[count(1)])
+- Exchange hashpartitioning(url#13, category#40, 200)
   +- *HashAggregate(keys=[url#13, category#40], functions=[partial_count(1)])
      +- *Project [url#13, category#40]
         +- *BroadcastHashJoin [url#13], [url#39], Inner, BuildRight
            :- *Project [url#13]
            :  +- *Filter isnotnull(url#13)
            :     +- *FileScan csv [url#13] Batched: false, Format: CSV, Location: InMemoryFileIndex
[file:/home/scala/visits.csv], PartitionFilters: [], PushedFilters: [IsNotNull(url)], ReadSchema: st
ruct<url:string>
            +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true]))
               +- *Project [url#39, category#40]
                  +- *Filter isnotnull(url#39)
                     +- *FileScan csv [url#39,category#40] Batched: false, Format: CSV, Location: In
MemoryFileIndex[file:/home/scala/urlinfo.csv], PartitionFilters: [], PushedFilters: [IsNotNull(url)]
, ReadSchema: struct<url:string,category:string>

scala> _
```

- Broadcast operations

  ➔ Shuffle data between nodes in cluster

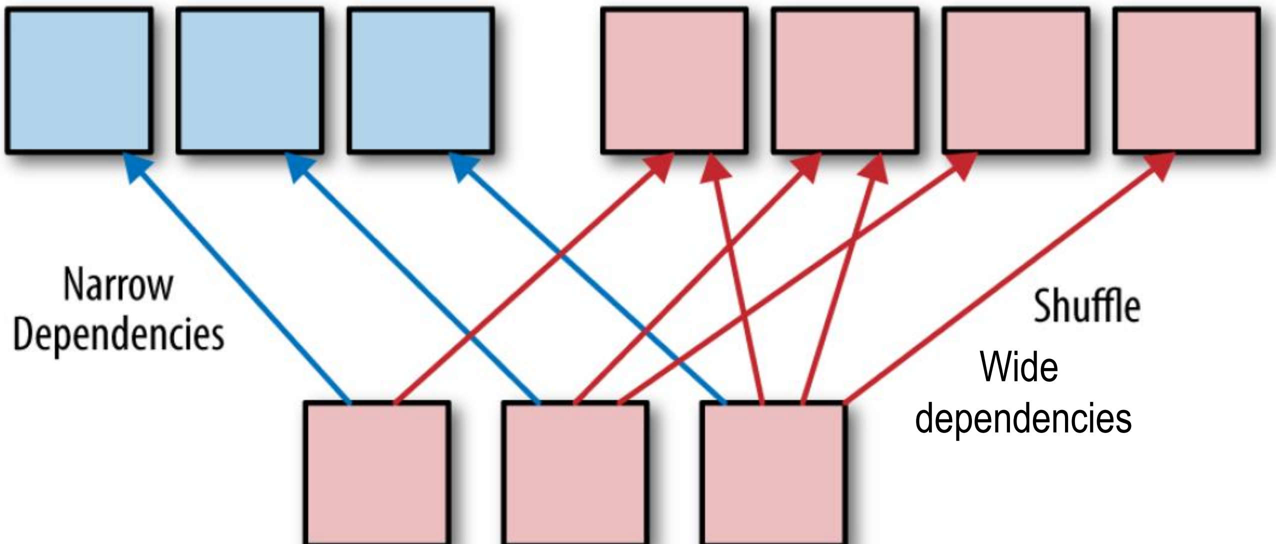
Default Join Method: Shuffle Join

- Remember: Partitioning method assigns a partition to a data object
  - Hash, range, or custom partitioners can be chosen in Spark
- If both RDDs use different custom partitioners (semantics is unknown for Spark), data has to be shuffled with a common partitioner



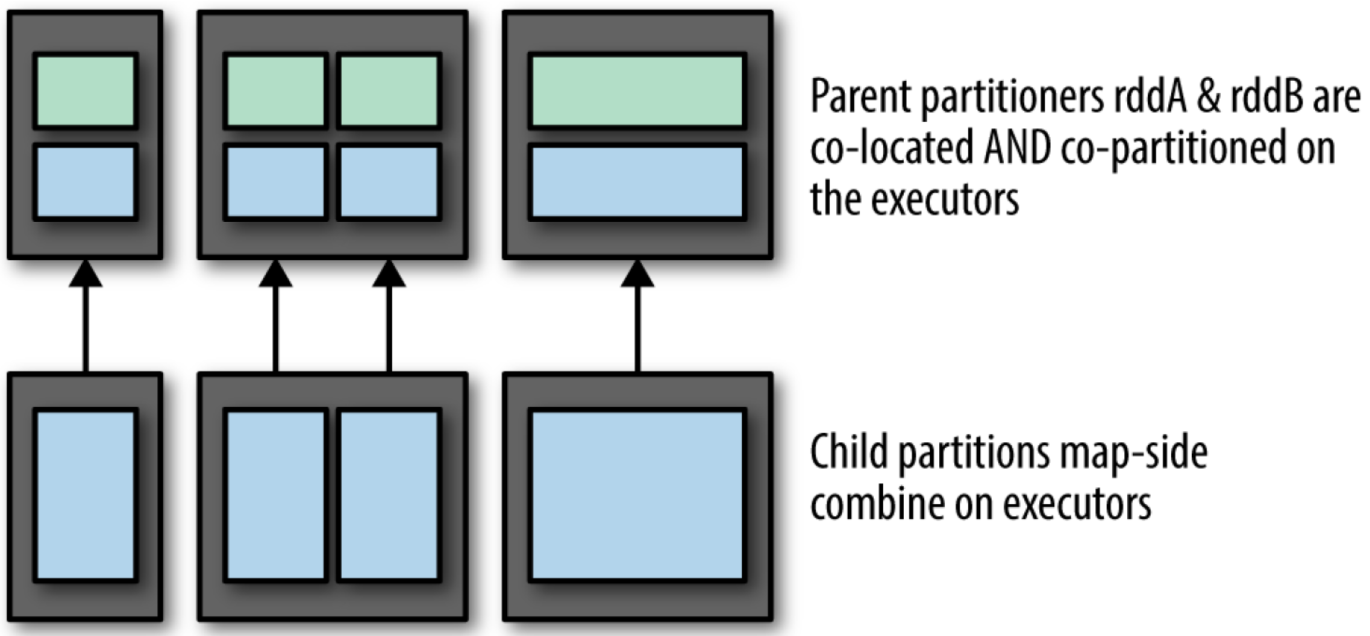
Join with Shuffle for only one RDD

- If RDD A has a known partitioner, RDD B does not, then only RDD B needs to be shuffled with the partitioner of RDD A
- Narrow dependencies: a partition depends only on one or two „parent" partitions
- Wide dependencies: a partition depends on multiple parent partitions



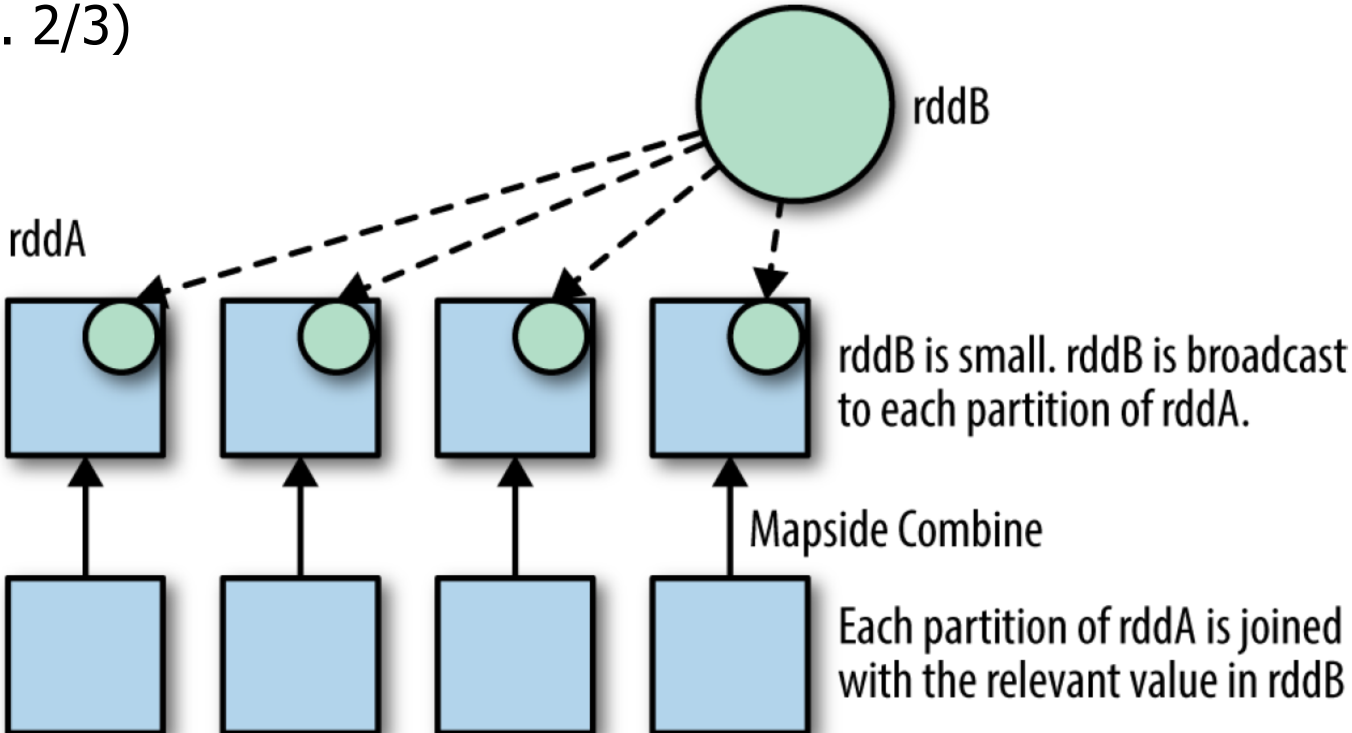
Co-located Join

- If both RDDs have the same partitioner, then the partitions to be joined are on the same node ➔ join is executed on these nodes



Broadcast Hash Join

- No common partitioner for RDDs
- Smaller RDD is sent to each worker node
- In the best case, this RDD fits into memory
  - Retain only those columns of the RDD which are necessary for the join (like semi-join, index-only query, optimized nested loop join ➔ Ch. 2/3)



Motivation (1)

- Some queries cannot be expressed in SQL or relational algebra
  - Give me a list of all parts that are required to build the component X.
  - Give me a list of all known ancestors of „John Doe".
- Recursion is required to express such queries
- Datalog is a language that allows recursion
  - Note: SQL-99 also includes recursion, but it has not been adopted to all DB systems
- Deductive DBs and their languages are the foundation for database theory

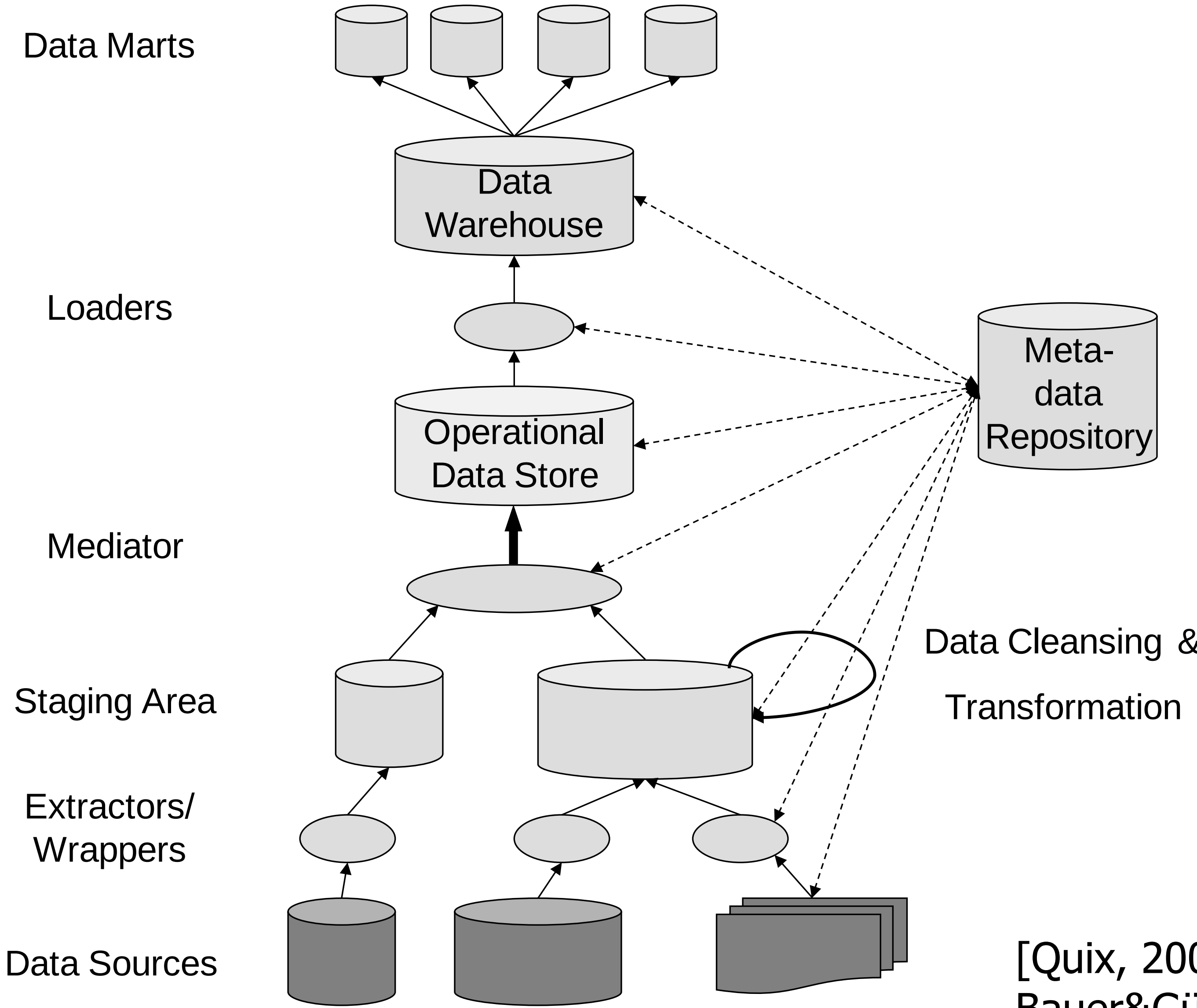
Motivation (2) - Data Warehouses (DWH)

- Organizations use DWHs to analyze current and historical data to identify useful patterns and support business strategies

- Emphasis is on complex, interactive, exploratory analysis of very large datasets created by integrating data from across all parts of an enterprise; data is fairly static

➡ Data Warehouses
  - Historical data for Decision Support
  - Long-running transactions
  - **Huge** amount of data, integrated from various sources
  - Focus on queries rather than updates
  - Mappings between data sources and data warehouse can be expressed as logical rules

Data Warehouse Architecture



[Quix, 2003; Bauer&Günzel, 2013]

Remember: Domain Relational Calculus (see Chapter 2.1.4)

- Atomic formulas are $r(X_1,\ldots,X_k)$ with r being a k-ary relation and $X_1,\ldots,X_k$ being variables or constants
- A database can be represented in the same way
  - A database is a set of *facts*
  - A *fact* is an atomic relation predicate only with constants representing one tuple of a relation, e.g.

    empl(122,'Miller', 'single',35.000,4)
    empl(101,'Meyer', 'married',55.000,4)
    dept(4,'computer',101)
    office(2,6232, 122)
    ...

DB Schema:
EMPL(eno, name, marstat, salary, dno)
DEPT(dno, dname, mgr)
Integrity constraints:
EMPL[dno] ⊆ DEPT[dno]
DEPT[mgr] ⊆ EMPL[eno]

Rules and Queries in Deductive DBs

- Rules: Horn clauses

$$p \text{ :- } r_1 \wedge r_2 \wedge \dots \wedge r_k$$

where p and r_i are relation predicates. e.g.

$p(X,Y)$:- $r_1(X,Z) \wedge r_2(Z,Y)$
manager(SSN,N) :- empl(SSN,N,M,S,D) $\wedge$ dept(D,DN,SSN)

Semantics: p(X,Y) is true if there exist some X,Y,Z such that every r_i is true

- Queries
  - Pure form with "?-", e.g. ?- empl(SSN,N,_,_,D) $\wedge$ dept(D,'computer',_).
  - Usually, use of special query predicate, e.g. $q(X,Y)$:- $r_1(X,Z) \wedge r_2(Z,Y)$

- Constraints
  - Rules without head that must be always false, e.g.

:- empl(S,N,M,Salary,D) $\wedge$ Salary<10.000

More expressive constraints than PKs and FKs are possible, similar to Assertions in SQL

Syntax & Terminology

- Syntactic conventions
  - Predicate names start with a lower case character
  - Variable names start with an upper case character
  - String atoms are enclosed in 'single quotes'
- Terminology

$$\overbrace{p(X,Y)}^{\text{Head (Consequence)}} \underbrace{:-}_{\text{Implication}} \overbrace{r_1(X,Z) \land r_2(Z,Y)}^{\text{Body (Condition)}}$$

  - *Distinguished variables:* appear in head and body, e.g. X and Y (∀)
  - *Non-distinguished (existential) variables:* appear only in body, e.g. Z (∃)
  - *Anonymous variable:* _
- Datalog vs. Prolog
  - Prolog allows function symbols as arguments of a predicate, Datalog does not

Examples

- Define rules for the Tax50 from chapter 3:

```
CREATE VIEW TAX50 AS SELECT e.* FROM EMPL e
 WHERE (e.marstat='single' AND e.salary<40.000)
 OR (e.marstat='married' AND e.salary<80.000);
```

- Constraint: an employee must not earn more than his/her manager

Model-theoretic approach

- Theory: schema (S) + integrity constraints (IC)
- Interpretation: database state
- Queries search through the interpretation
- Modifications must take the theory into consideration

Proof-theoretic approach

- Theory: Facts and deduction rules (T) +
  integrity constraints (IC)

- Queries are theories, which must be proved by using the axioms T.

- A modification adopts a new theory (e.g., T') and it should not violate the integrity constraints, i.e.

$$T' \models IC$$

Example

T: empl(12, 'jim', 50000, 2).
empl(11, 'jones', 60000, 2).
dept(2, 'R&D', 11).

works_dir_for(X,Y) :- empl(_,X,_,D), dept(D,_,M), empl(M,Y,_,_).

works_for(X,Y) :- works_dir_for(X,Y).
works_for(X,Y) :- works_dir_for(X,Z), works_for(Z,Y).

same_manager(X,Y) :- works_for(X,M), works_for(Y,M), X<>Y.

IC: :- empl(X,_,S,_), S<10000 .
:- empl(X,_,S,_), S>90000 .

Query: ? - works_for(X, jones).

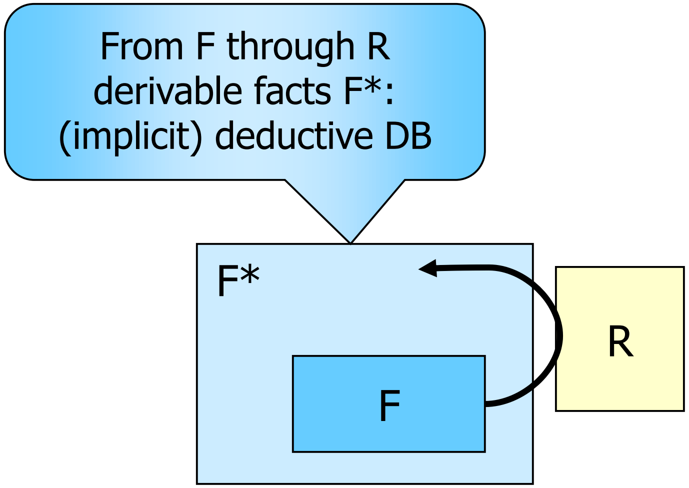
DB Schema:
EMPL(eno, name, salary, dno)
DEPT(dno, dname, mgr)
Integrity constraints:
EMPL[dno] $\subseteq$ DEPT[dno]
DEPT[mgr] $\subseteq$ EMPL[eno]

Definition of DDBs

Definition 5.1:

A *deductive database* consists of a set F of facts, a set R of deduction rules, a set IC of integrity constraints and the set F* of all explicit and derived (implicit) facts.

- EDB: extensional DB
  - Relations defined as a set of facts in F
  - Base relations
  - Set of facts



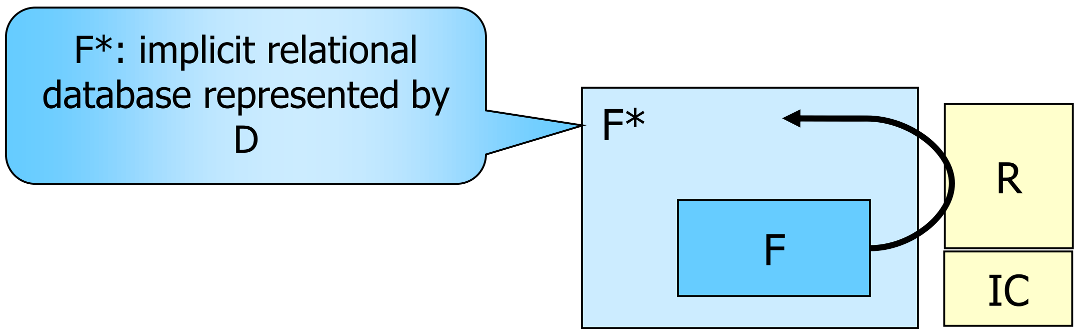
- IDB: intensional DB
  - Relations defined by rules in R
  - Derived relations

Datalog and its variations

- Datalog$_\neg$: Negation is allowed
- NR-Datalog: Recursion is not allowed

Semantics of Deductive Databases

Deductive database D=(F, R, IC)



Question:

- What is the formal definition of F*?
- Is F* uniquely determinable?
- What is the meaning of "derivable"?

Problems are caused by negation and recursion, therefore the classes NR-DATALOG, NR-DATALOG$^{\neg}$ and DATALOG$^{\neg}$ have to be considered separately.

Semantics of DDBs: NR-Datalog

F* is formally defined as the minimal *Herbrand model* of D.

Definition 5.2 (Herbrand Base of D):
All positive ground literals are constructable from predicates in D and constants in D.

Definition 5.3 (Herbrand Model of D):

A *Herbrand Model* is every subset M of the Herbrand Base of D, such that:

Every fact from F is contained in M. For every ground instance of a rule in D over constants in D, if M contains all literals in the body, then M contains the head as well.

A minimal model does not properly contain any other model.

Example: NR-DATALOG

F: q(a,b) t(b) R: $p(X) \leftarrow q(X,Y), t(Y)$
q(b,a)

- The Herbrand base is

q(a,a)	q(a,b)	p(a)	t(a)
q(b,a)	q(b,b)	p(b)	t(b)

- The Herbrand models M_i are consequently

Minimal

M_1: q(a,b) t(b)
q(b,a) p(a)

and

M_2: q(a,b) t(b) p(a)
q(b,a) p(b)
q(b,b)

- The ground instances of the rule are thus (blue = contained in M_i)

p(a)	←	q(a,a), t(a)
p(a)	←	q(a,b), t(b)
p(b)	←	q(b,a), t(a)
p(b)	←	q(b,b), t(b)

and

p(a)	←	q(a,a), t(a)
p(a)	←	q(a,b), t(b)
p(b)	←	q(b,a), t(a)
p(b)	←	q(b,b), t(b)

Least Fixpoint for NR-Datalog

F* is created by the repeated (finite) application of the immediate consequence operator T_D (naive evaluation strategies) starting from F results in derivation of implicit facts from F:

$$T_D(T_D(....(T_D(F))...))$$

Definition 5.4:
For a subset S of a Herbrand Base the application of T_D to S is defined as:

$T_D(S) := \{ H : (H \leftarrow B)$ is a ground instance of a rule such that S contains all literals in B$\}$

Start by F		(=F0)
Add $T_D(F_0)$	$F_0 \cup T_D(F_0)$	$(=F_1)$
Add $T_D(F_1)$	$F_1 \cup T_D(F_1)$	$(=F_2)$
		↓
		F*

Operator correspond to this construction
$T_D{}^*(S) = S \cup T_D(S)$

Least fixpoint of this operator: minimal solution is $X = T_D{}^*(X)$

Theorem 5.1: (v. Emden, Kowalski 1976)

The uniquely determined least fix point of $T_D{}^*$ is the minimal Herbrand Model of D

Compute the least fixpoint!

q(X) :- r(X,Y), NOT b(X).
r(X,Y) :- c(X,Y), b(Y).
r(X,Y) :- c(X,Z), r(Z,Y).

c(1,2).
c(2,3).
c(1,4).

b(3).
b(4).

Semantics of DDBs: NR-DATALOG$^{\neg}$

Problem: If negation occurs in the body of a rule, then there may be <u>no unique</u> minimal Herbrand model any more. Which one is the "natural model" of D, and thus represents the intended semantics of D?

Example:

F: q(a,b)
q(b,a) t(b)

R: $p(X) \leftarrow q(X,Y)$, NOT $t(Y)$

- The minimal Herbrand models are
  M_1: $F \cup \{p(b)\}$
  M_2: $F \cup \{t(a)\}$

p(b) is derivable from F, M_1 is the "natural" model of D

No reason why t(a) should be true.

- Ground instances of the rules (in M_1, in M_2, in M_1 and M_2):
  $p(a) \leftarrow q(a,a)$, NOT $t(a)$
  $p(a) \leftarrow q(a,b)$, NOT $t(b)$
  $p(b) \leftarrow q(b,a)$, NOT $t(a)$
  $p(b) \leftarrow q(b,b)$, NOT $t(b)$
- Main problem in presence of negation: characterization of the "natural" model of D (representing the intended semantics of D).

Example (NR-DATALOG$^\neg$)

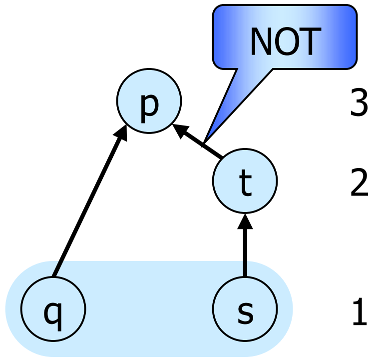
Least fixpoint characterization cannot be directly adopted

F: q(a,b)   s(b)   R: p(X) $\leftarrow$ q(X,Y), NOT t(Y)
     q(b,a)   t(Y) $\leftarrow$ s(Y)

- First application of T_D:
  - No t-fact in F $\Rightarrow$ p(a) and p(b) are derivable
  - s(b) in F $\Rightarrow$ t(b) is derivable
- Second application of T_D:
  - No new fact derivable $\Rightarrow$ fixpoint reached
- It follows therefore:
  - least fixpoint of T_D^*: $F \cup \{t(b), p(a), p(b)\} \neq$
  - "natural" Herbrand model: $F \cup \{t(b), p(b)\}$
- Reason (intuitively): "t(b) arrives too late for preventing derivation of p(a)."

Stratification

Predicates "call" each other in a hierarchical order:



⇒ Predicates can be *layered* (or: *stratified*); no two predicates in a layer (*stratum*) depend negatively on each other. If a predicate p depends on a negative predicate r, then r is in a lower layer.

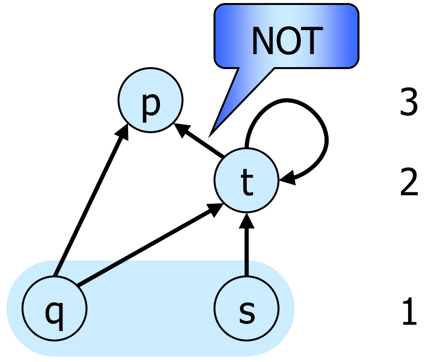
- If application of T_D is done layer by layer, the least fixpoint of D is consequently the *natural* Herbrand model.

- It follows therefore:
  1. layer: F
  2. layer: $F \cup \{t(b)\}$
  3. layer: $F \cup \{t(b)\} \cup \{p(b)\}$

Problem: The recursion leads possibly to more than one application of T_D per layer. If negations do not occur in the recursion cycle, there will be no problem.

Example:

F:	q(b,c)	s(c)	R:	p(X)	$\leftarrow$ q(X,Y), NOT t(Y)
	q(b,a)			t(Y)	$\leftarrow$ q(Y,Z), t(Z)
				t(Y)	$\leftarrow$ s(Y)



- From this it follows:
  1. layer: F
  2. layer: F $\cup$ {t(c)}
     F $\cup$ {t(c)} $\cup$ {t(b)}
  3. layer: F $\cup$ {t(c)} $\cup$ {t(b)} $\cup$ {p(b)}

....

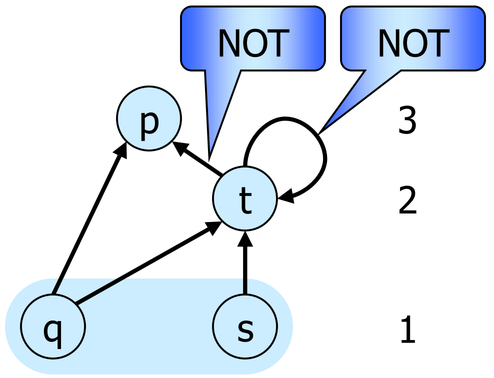
Non-stratified DATALOG$^{\neg}$ Programs

Problem: Negation in the recursion cycle violates the stratification condition.

Example:

F:	q(a,b)	s(c)	R:	p(X)	← q(X,Y), NOT t(Y)
	q(b,a)			t(Y)	← q(Y,Z), NOT t(Z)
				t(Y)	← s(Y)

- Even when applying T_D for t-rules only results in „anomalies":
  - t(a) is generated, as t(b) is initially missing
  - t(b) is generated, as t(a) is initially missing



- Do t(a) and t(b) belong to the "natural" Herbrand model now? Is there any reasonable "natural" Herbrand model at all?

Example (cont.)

- t(a) and t(b) are not "natural" consequences of R ∪ F:

  $$t(a) \leftarrow q(a,\underset{b}{Z}),\ \text{NOT}\ t(\underset{b}{Z})$$

  $$t(b) \leftarrow q(b,\underset{a}{Z}),\ \text{NOT}\ t(\underset{a}{Z})$$

- Consequently it results in two exclusive cases:

  either NOT t(b), consequently t(a)

  or NOT t(a), consequently t(b).

- So that it yields also two different models:

  F ∪ {p(a), t(a)} , also NOT t(b)

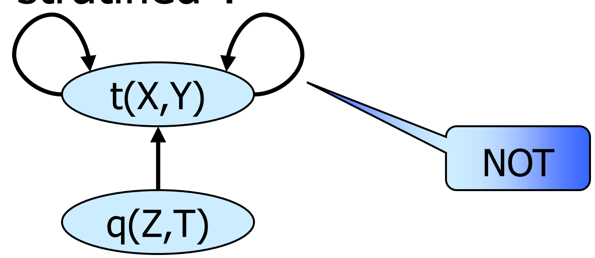
  or F ∪ {p(b), t(b)} , also NOT t(a).

Local Stratification

Example DATALOG$^{\neg}$ [cont.]:

F:	q(a,b)	R:	t(X,Y)	← q(X,Y), NOT t(Y,X)
	q(b,c)		t(X,Y)	← t(X,Z), t(Z,Y), NOT t(Y,X)

R is not stratified, but “locally stratified”:



“Reachable” instances of R:

t(a,b)	← q(a,b), NOT t(b,a)
t(b,c)	← q(b,c), NOT t(c,b)
t(a,c)	← t(a,b), t(b,c), NOT t(c,a)

- No recursion cycle involves negation among “reachable” instances.
- Test for local stratification expensive, but can be computed “on the fly”, i.e., during evaluation of the rules without computing the (local) stratification [Bry, 1989]

Are these programs stratified?

Program 1

q(X) :- NOT p(X), t(X).
p(X) :- s(X,X), NOT r(X).
s(X,Y) :- s(Y,X), t(Y).
r(X) :- t(X), NOT s(X,X).

Program 2

p(X) :- q(X,Y), NOT t(Y).
t(Y) :- q(Y,Z), NOT t(Z).
t(Y) :- s(Y).

Bottom-Up vs. Top-Down Evaluation

- **Bottom-Up (Forward Chaining):**
  - Generation of implicit facts at evaluation-time
  - Evaluation of the query against temporarily materialized implicit databases.
    (direct implementation of least fixpoint computation)
  - **Drawback:** when materializing F*, the particular query Q is not considered → many irrelevant answers and intermediate results may be generated.

- **Top-Down (Backward Evaluation):**
  - Generation of subqueries until queries to base relations are reached
  - Evaluation of base subqueries against F and upward propagation of answers to the top query
  - As opposed to bottom-up approach:
    constants in top query and subqueries are passed downwards and provide restrictions while evaluating base queries.
  - **Drawback:** Inefficient (or not terminating) for recursive queries

Example: Top-Down Evaluation

F: q(a,b) q(a,c) t(d)
q(b,c) q(b,d)
q(c,a)

R:
p(X,Y) ← q(X,Z), r(Z,Y)
r(Z,Y) ← q(Z,Y), NOT t(Y)

Q: p(a,Y)

Match query with head of 1st rule and generate subqueries for body:

⟹ q(a,Z), r(Z,Y)

q(a,Z) can be proven with facts from F, e.g., take q(a,c)

Choice Point

⟹ q(a,c), r(c,Y)

prove r(c,Y) by using second rule

⟹ q(c,Y), NOT t(Y)

q(c,Y) can be proven by q(c,a) in F

⟹ q(c,a), NOT t(a)

NOT t(a) can be also proven by F, thus, r(c,a) and p(a,c) are true

⟹ derivation of one result finished: p(a,c)

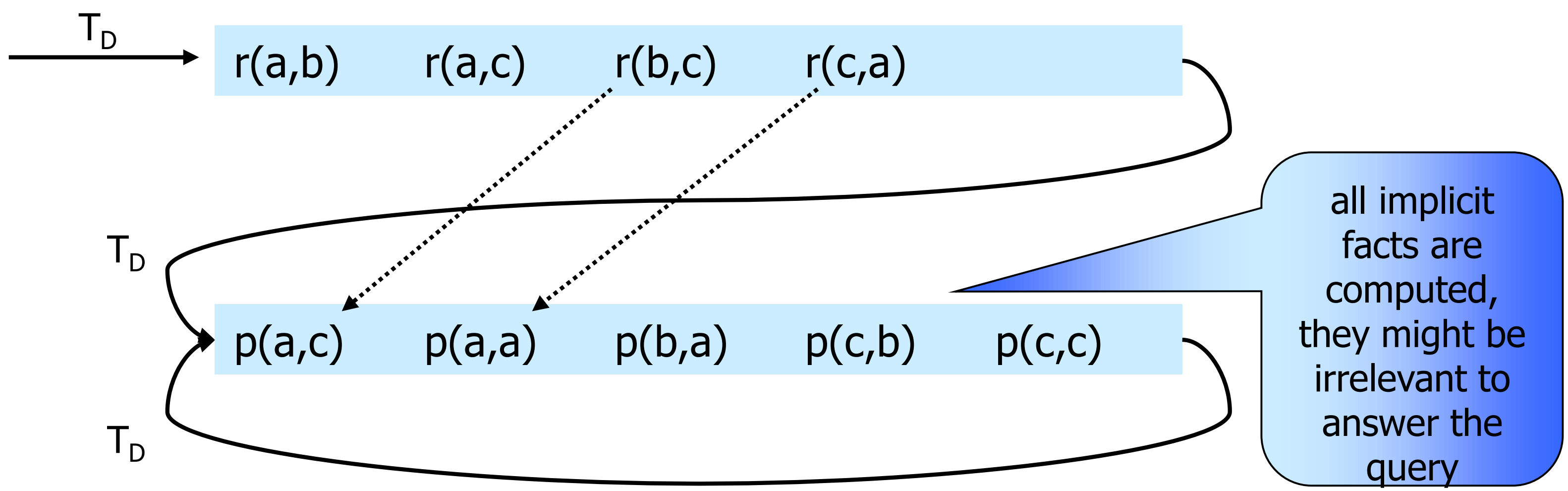
⟹ backtrack to choice point and generate next result (if required)

Example: Bottom-Up Evaluation

F: q(a,b) q(a,c) t(d)
q(b,c) q(b,d)
q(c,a)

R:
p(X,Y) ← q(X,Z), r(Z,Y)
r(Z,Y) ← q(Z,Y), NOT t(Y)

F*:



Q: p(a,Y)

Example: Bottom-Up Evaluation

F:	q(a,b)	q(a,c)	t(d)	R:	p(X,Y) ← q(X,Z), r(Z,Y)
	q(b,c)	q(b,d)			r(Z,Y) ← q(Z,Y), NOT t(Y)
	q(c,a)				

$\xrightarrow{T_D}$ r(a,b) r(a,c) r(b,c) r(c,a)

F*: T_D

p(a,c) p(a,a) p(b,a) p(c,b) p(c,c)

T_D

all implicit facts are computed, they might be irrelevant to answer the query

Note: Identical computation for different queries!

Q: p(X,a)

Integrity Constraints

Definition 5.5:
Integrity constraints (IC) are conditions that have to be satisfied by a database at any point in time (expressing general laws which cannot be used as derivation rules) .

Definition 5.6:
Integrity-checking tests, whether a particular update is going to violate any constraint.

- **Main problem for IC-tests**:
  Full evaluation of all ICs before every update would be very expensive and would decrease update performance significantly.
- **Solution**:
  Determine a reduced set of **simplified ICs** for which the checking guarantees satisfaction of **all ICs**.
- This approach leads to a **specialization of constraints**.

Representation of Constraints and Updates

- Integrity constraints expressed as **Datalog**$^{\neg}$ rules for a meta predicate "inconsistent".

- Updates are either **atomic** insertions / deletions of facts or **general** updates (depending on a condition, which is a Datalog query to be evaluated before the update).

Example:

inconsistent ← employee(X), works_for(X,X)
(original constraint: no employee works for himself)

insert works_for(john, jim)
delete employee(X) where works_for(X, john)

Constraint Specialization

Input:

- Update U={delete L, insert L}
- Integrity constraints IC (satisfied before the update)

1. IC is affected by U, if IC contains a literal L* that is unifiable with L (resp. NOT L), if U is an insertion (resp. deletion).

2. For every such L*, IC_σ is a relevant instance of IC with respect to U. (where σ is a most-general-unifier of L and L*)

3. Simplified relevant instances of IC with respect to U are obtained by deleting L* from relevant instances.

So far, generalized updates, derivation rules and negations have not been taken into account yet.

Example: Constraint Specialization

IC: inconsistent ← p(X,Y), NOT s(X) (corresponding: FORALL X,Y: p(X,Y) ⇒ s(X))

- insert p(a,b) — IC affected!
  inconsistent ← p(a,b), NOT s(a)

- delete p(a,b)

- insert s(a)

- delete s(a) — IC affected!
  inconsistent ←p(a,Y), NOT s(a)

Generalized Updates

Example:

IC: inconsistent ← p(X,Y), NOT s(X)

- Insert p(X,b) where r(X)

1. IC is affected by U, if IC contains a literal L* that is unifiable with L (resp. NOT L), if U is an insertion (resp. deletion).
2. For every such L* IC_σ is a relevant instance of IC regarding U.
3. Simplified relevant instances of IC regarding U are obtained by deleting L* from relevant instances.

Instead of a simplified relevant instance, we end up with a specialized constraint:

inconsistent ← r(X), NOT s(X)

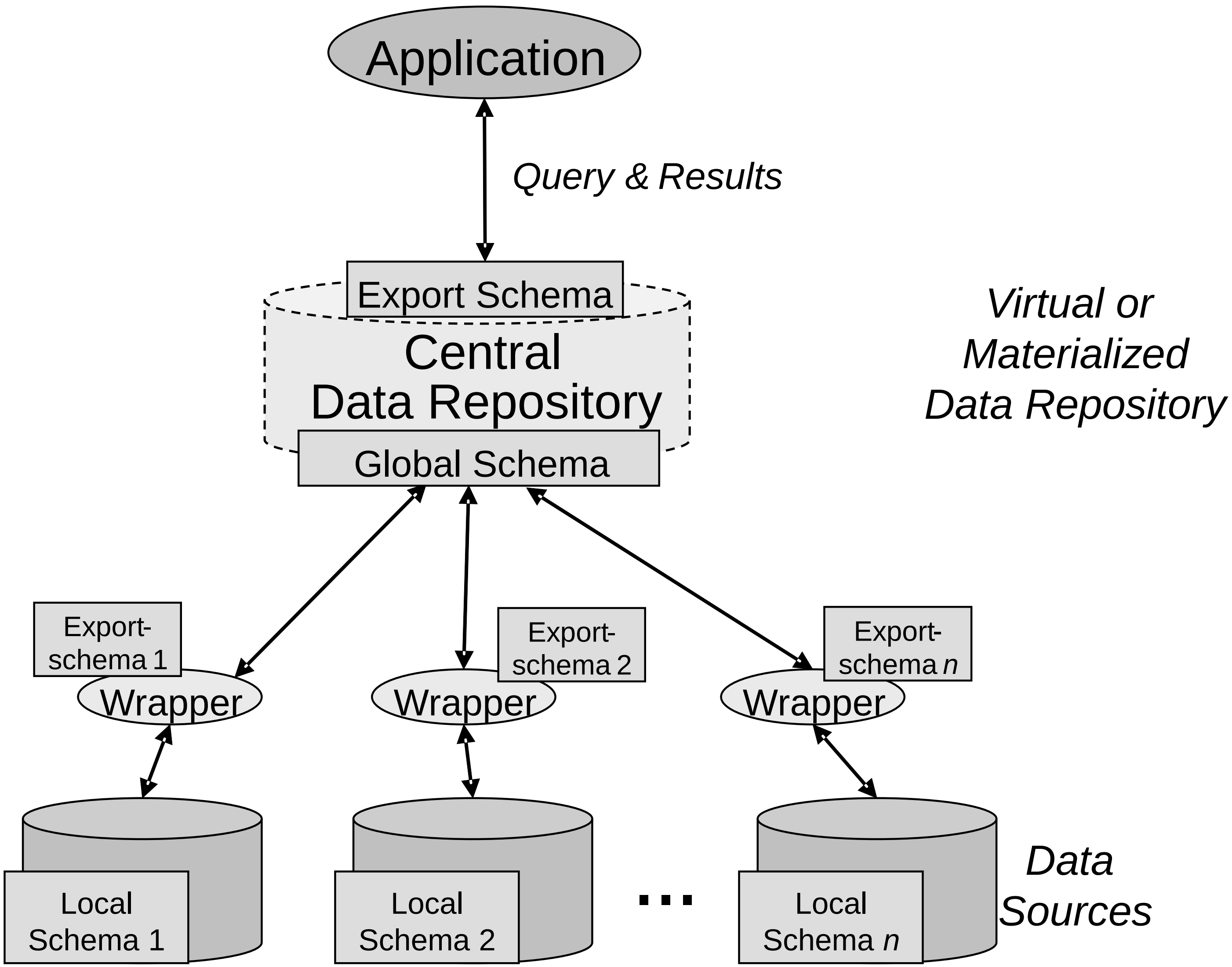
5.3 Queries in Data Integration Systems

Definition 5.7:
Data integration is the problem of providing unified and transparent access to a collection of data stored in multiple, autonomous, and heterogeneous data sources

Motivations for data integration

- Company mergers
- Reorganization of companies
- Restructuring of databases in an organization
- Combination of data from several internal and external sources for data analysis (e.g. data warehousing, OLAP)

General Integration Architecture



[Quix, 2003]

Data Integration Approaches

- Distributed databases
  - Data sources are homogeneous DBs under central control
- Multidatabase or federated databases
  - Data sources are autonomous, heterogeneous databases
- **(Mediator-based) Data Integration**
  - Access through a global schema mapped to autonomous and heterogeneous data sources
- Data Exchange
  - Mapping between a source and a target system
  - source may specify data in target only partially
- Peer-to-peer data integration:
  - Network of autonomous systems mapped one to each other, without a global schema

Challenges

- Modeling of the global schema, the sources, and **the mappings between the two**
  - Construction of the global schema
  - Discovering mappings between sources and global schema
- Legacy databases: DBs are used for many applications, structure cannot be changed.
- Heterogeneity & Conflicts
  - Data Types & Structures
  - Semantics: Meaning of terms on schema- and instance-level
- **Answering queries expressed on the global schema**
- Data extraction, cleaning, and reconciliation
- Processing of updates expressed on the global schema and/or the sources ("read/write" vs. "read-only" data integration)
- Optimization of queries across data sources

Querying Problem

- A query expressed in terms of the global schema must be reformulated in terms of (a set of) queries over the sources and/or materialized views
- The computed sub-queries are shipped to the sources, and the results are collected and assembled into the final answer
- The computed query plan should guarantee:
  - completeness of the obtained answers wrt the semantics
  - efficiency of the whole query answering process
  - efficiency in accessing sources

Mappings

How to specify the mapping between the data sources and the global schema?

- **LAV (local-as-view, source-centric):**
  The sources are defined in terms of the global schema (i.e. as view on the global schema)

- **GAV (global-as-view, global-schema-centric):**
  The global schema is defined in terms of the sources (i.e. as view on the source schemas)

- **GLAV (combination of GAV and LAV)**

Example

- Source Schema S
  - em50(Title, Year, Director)
    European movies since 1950
  - rv10(Movie, Review)
    reviews since 2000

- Global Schema G
  - movie(Title, Director, Year)
  - ed(Name, Country,Dob) *(European directors)*
  - rv(Movie, Review)

- Query q

```
SELECT M.title, R.review
FROM Movie M, RV R
WHERE M.title=R.title AND M.year = 2000
```

Questions

- Can we rewrite q as a query over the source schema S?
- Is there a unique solution?
- If not, can we characterize a best solution?
- What is the semantics of a query?
- And how do we specify the mapping between S and G?

Problem: Expressivity

- In order to answer such questions, we have to prove that queries are equivalent or contained in each other
  - Query Equivalence
    - q and q' are equivalent if they produce the same result for all legal databases
  - Query Containment
    - q is contained in q' if the result of q is a subset of the result for q' for all legal databases
- Undecidable for many query languages
- Restriction to conjunctive queries
  ➔ Tableau Method

GAV Example

- movie(Title, Year, Director) :-
  em50(Title, Year, Director).

- ed(Name) :-
  em50(_,_,Name).

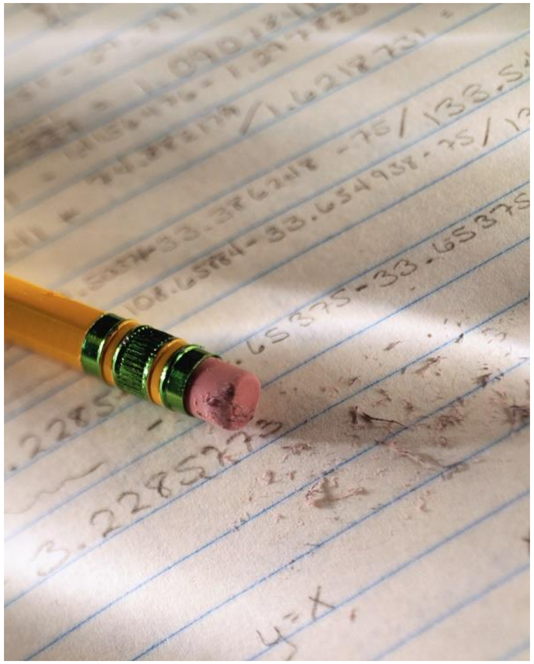
- rv(Movie, Review) :-
  rv10(Movie,Review).

Query Rewriting in GAV

Queries over $\mathcal{G}$ can be rewritten as queries over $\mathcal{S}$ by unfolding

q(Title, Review) :-
movie(Title, 2000,_),
rv(Title,Review).

➡ q'(Title, Review) :-
em50(Title, 2000,_),
rv10(Title,Review).



LAV Example

- em50(Title,Year,Director) :-
  movie(Title,Year,Director),
  ed(Director,Country,Dob),
  Year ≥ 1950.

- rv10(Movie,Review) :-
  rv(Movie,Review),
  movie(Movie,Year,Director),
  Year ≥ 2000.

Query Rewriting in LAV

- Sources are views
- Answer queries on the basis of available data in the views
- Research area: Answering queries using views

➔ **Query Optimization:** Answer a query using a materialized view!

- **Idea:** Try to cover the predicates of the query by predicates in the bodies of the source views

Query Rewriting in LAV Example

- q(Title, Review) :-
  movie(Title,Year,_),
  rv(Title,Review),
  Year = 2000.

- em50(Title,Year,Director) :-
  movie(Title,Year,Director),
  ed(Director,Country,Dob),
  Year ≥ 1950.

- rv10(Movie,Review) :-
  rv(Movie,Review),
  movie(Movie,Year,Director),
  Year ≥ 2000.

Summary

- Query Processing in Big Data
  - Data partitions & replications enable parallel, distributed, fault-tolerant query processing
  - Map-Reduce is a generic programming pattern for distributed computation, but often too rigid for complex data analysis
  - High-Level languages enable declarative query processing in distributed systems
- Deductive databases are the logical basis for relational databases
  - Herbrand model corresponds to least fixpoint of T_D
  - Negation and recursion require stratified computation of fixpoints
  - Integrity constraints are a special form of rules
- Application in Information Integration
  - Mappings between data sources and integrated schema can be represented as logical rules
  - Queries to integrated schema have to be rewritten into queries for data sources
  - Mappings can be specified as LAV or GAV mappings

Review Questions

- Explain the Map-Reduce programming pattern? What is done in the Map function, what is done in Reduce?
- What are the problems of the Map-Reduce programming pattern?
- What is the goal of systems like Pig Latin or Hive?
- What is a broadcast hash join in Spark?
- When do you need to do shuffling in Spark? What are narrow and wide dependencies?
- Sketch and explain the data warehouse architecture!
- What is the Herbrand Base and the Herbrand Model?
- How can you compute the minimal Herbrand Model?
- Translate a RA query into Datalog and vice versa!
- What is stratification?
- Explain top-down and bottom-up evaluation of Datalog programs!
- Show an example for a mapping between a source schema and an integrated schema using the Datalog notation.
- What is the difference of GAV and LAV mappings?

References

[Bauer & Günzel, 2013] Bauer, A. & Günzel, H. (ed.) Data-Warehouse-Systeme: Architektur, Entwicklung, Anwendung dpunkt-Verlag, 2001

[Bry, 1989] Bry, F. Logic programming as constructivism: A formalization and its application to databases Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, 1989, 34-50

[Gottlob et al, 2012] Ceri, S.; Gottlob, G. & Tanca, L. Logic programming and databases Springer Publishing Company, Incorporated, 2012

[Quix, 2003] Quix, C. Metadata Management for quality-oriented Information Logistics in Data Warehouse Systems (in German) RWTH Aachen University, 2003

[Schöning, 2000] Schöning, U. Logik für Informatiker Spektrum Akademischer Verlag, 2000