BAHIA – 06/11/2019

# Answering GPSJ Queries in a Polystore: a Dataspace-Based Approach

HAMDI BEN HAMADOU, ENRICO GALLINUCCI, AND MATTEO GOLFARELLI

Business Intelligence Group

# Goal

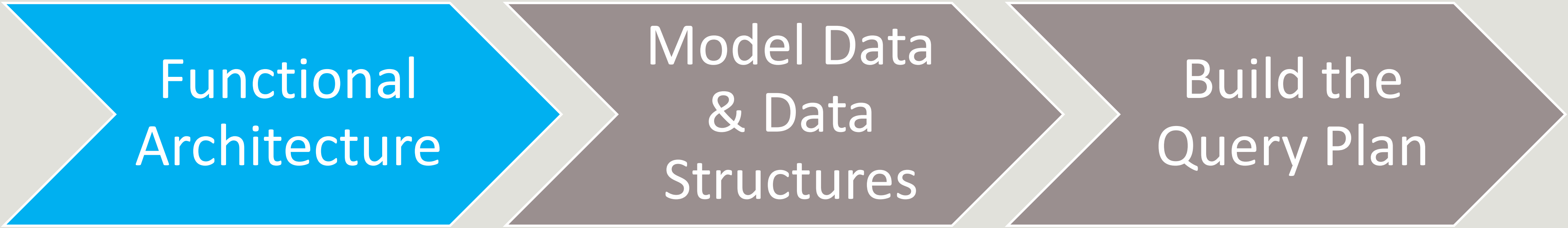
OLAP analyses traditionally run on Data Warehouses that :

- 👍 Ensure high-quality data through ETL processes that integrate and clean sources
- 👎 Are typically built on a fixed relational schema obtained through a *schema first – data later* approach
  - 👎 Not suitable for extemporaneous analysis as it requires a very high initial effort
  - 👎 Typically hard to evolve in case of a new source is added, or source schemata change

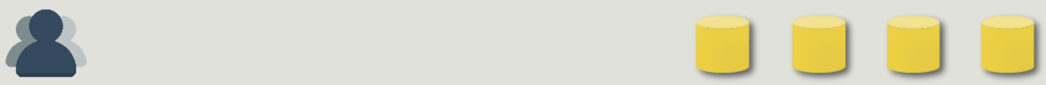
Our goal is to perform OLAP analyses beyond the boundaries of conventional Data Warehouses

- Adopt a more flexible and lightweight approach to data analysis
- Handle heterogeneous schemas and data models
- Adopt a pay-as-you-go integration approach where the integration is progressively carried out by the user as the available data is explored

# Presentation Layout



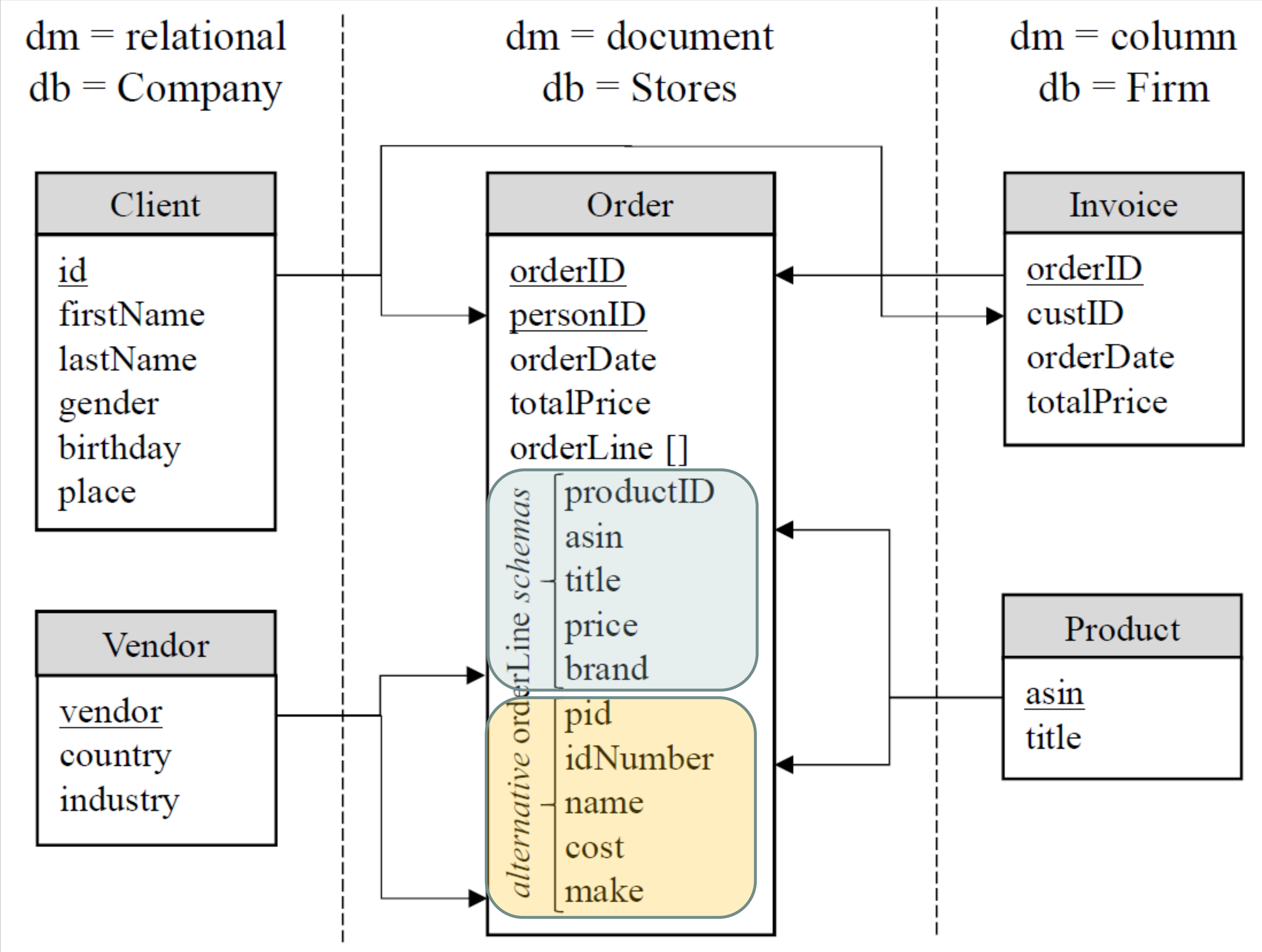
# Functional Architecture and Assumptions



**Types of heterogeneity**

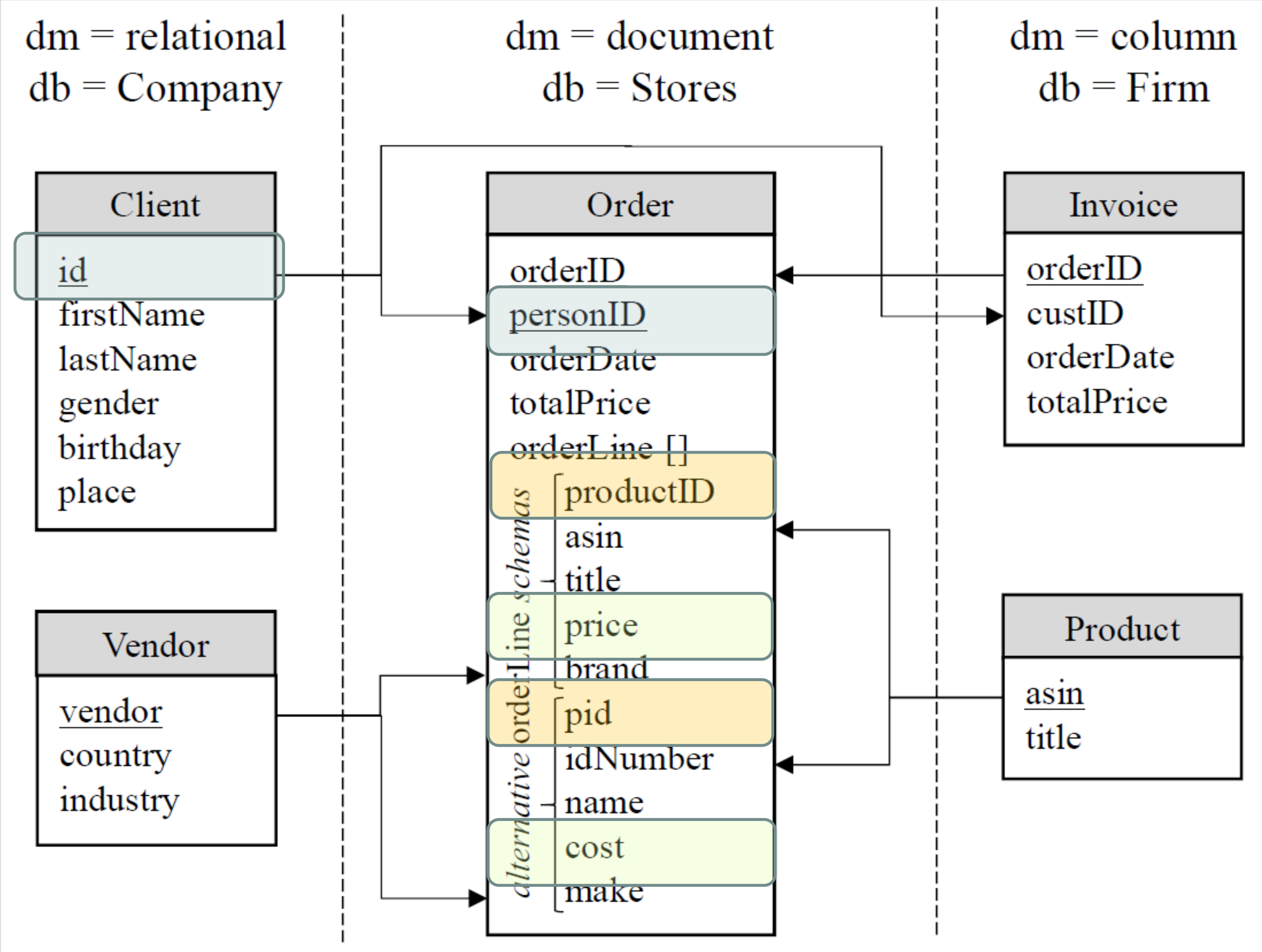
- **Schema**
  - Missing attributes
  - Different data types
  - Different naming convention
- **Data model**
  - Relational
  - Document
  - Column-based

# Heterogeneity




**Missing attributes**

# Heterogeneity



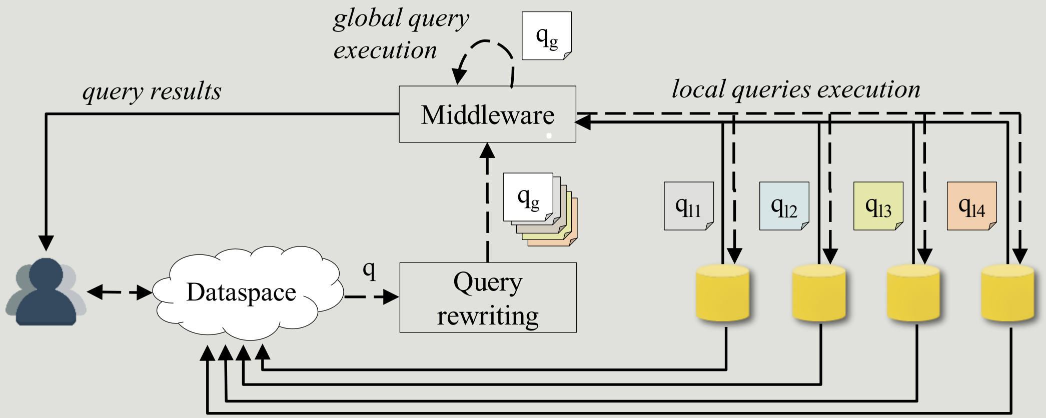

**Different Naming Conventions**

# Functional Architecture and Assumptions


Schema information are collected in a **data space** and eventually integrated in a **pay-as-you go approach** (the more integrated concepts you want, the more the effort you need)

A **dataspace** is a lightweight integration approach providing basic query expressive power on a variety of data sources, bypassing the complexity of traditional integration approaches and possibly returning best-effort or approximate answers

# Functional Architecture and Assumptions



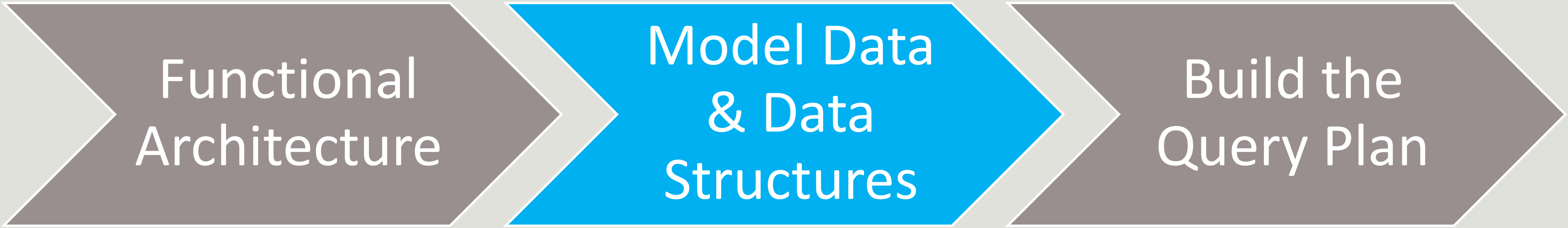

Queries expressed on the data space are:

1) rewritten on the single sources as a set of **local queries**

2) results are collected in a **central repository**

3) a **global query** further integrates local results so that a global result can be returned to the user

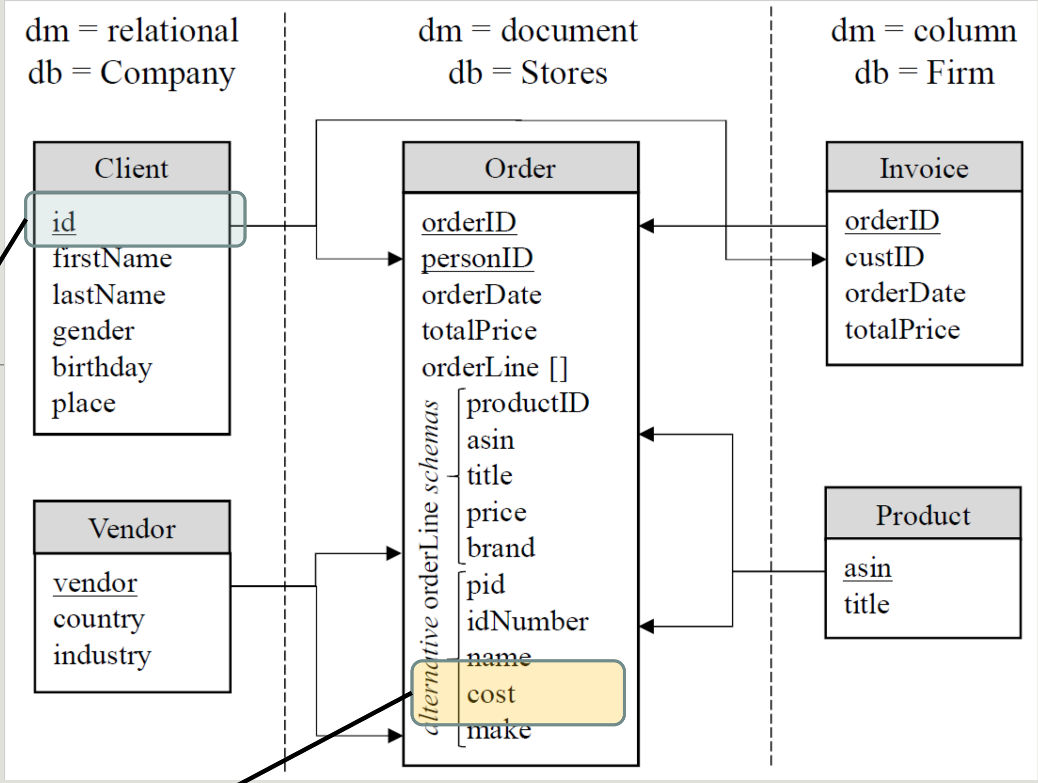
# Presentation Layout



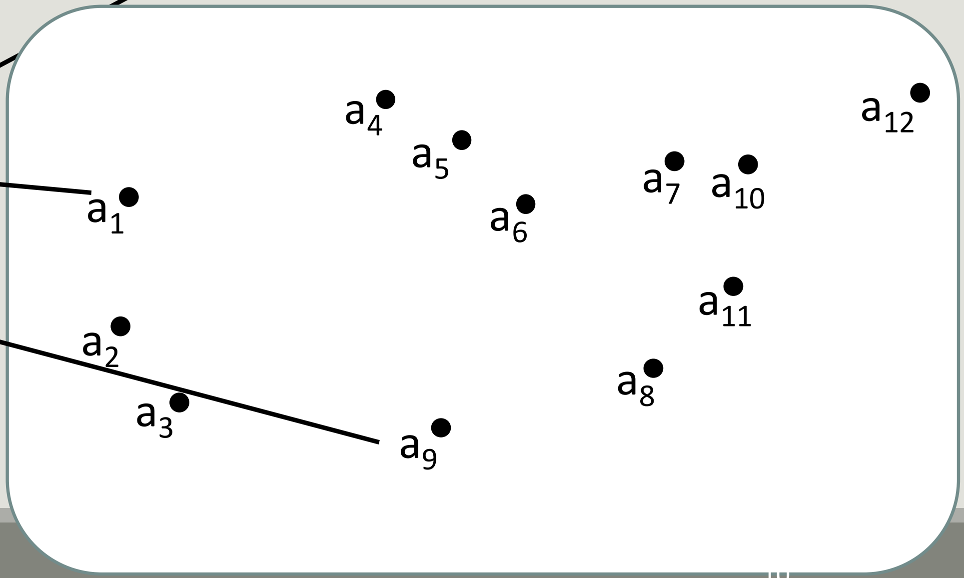
# Attributes

We define an **attribute** as **a=(dm, db, col, name)**

- **dm** = {relational|column|document} is the data model
- **db** $\in$ D is the database name
- **col** is the collection name in db
- **name** is the name of the attribute in the collection col




- $a_1$: (relational, Company,Client,id)
- $a_9$ : (document, Stores, Order, orderLine.cost)



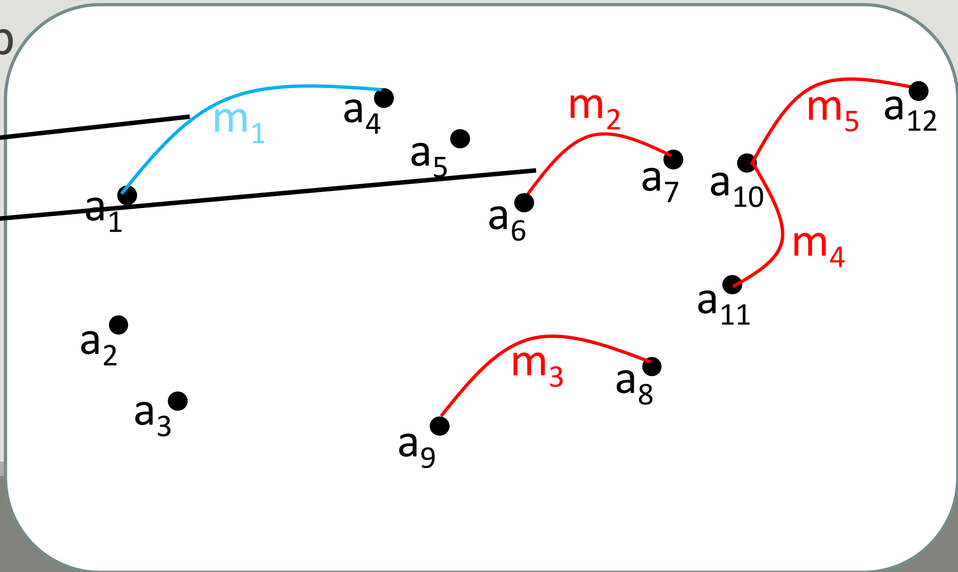

**In a polystore relations are not first citizen**

# Mapping & Features

Dataspace must be able to hide schema heterogeneity and to provide a global representation

A **mapping** is a relationship between two attributes
a' and a''. We define a mapping as m=(**a',a'', ϕ, φ,ψ**), where
**a',a''** $\in$ A*
ϕ is the type of the mapping [**sameAs** | **fk**]
φ is a transcoding function to express a' values in a'' format
ψ is the semantics describing the meaning of the relationship (limitedly to fk mappings).

- $m_1$: ($a_4$, $a_1$, fk, toInt(), *''client order''*)
- $m_2$ : ($a_6$, $a_7$, sameAs,I())

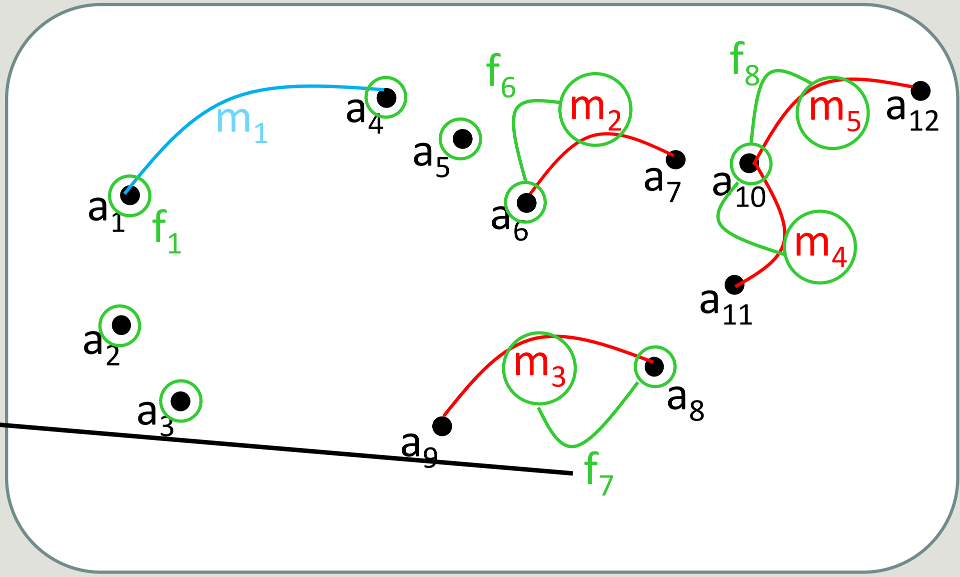


# Mapping & Features

Single mappings are not sufficient since there can be sets of attributes representing the same concept

We define a **feature** as f=(**name**,**a**,**M**), where
**name** is the name of the feature
**a** is the representative attribute of the feature
**M** is a set of sameAs mappings

- $f_1$: (id, $a_2$, $\emptyset$)
- ………….
- $f_7$: (orderline.price, $a_8$, $\{m_3\}$)
- $f_8$: (orderline.brand, $a_{10}$, $\{m_4, m_5\}$)




A **Dataspace** D is a set of feature

**Since in a dataspace features represents attribute, there is no notion of relations as first citizen**

# GPSJ queries

The query expressiveness that we consider covers a wide class of queries by composing three basic SQL operators: selection, join and generalized projection.

SELECT orderLine.ProductID, avg(orderLine.price)
FROM [….]
WHERE orderLine.Brand = 'ABC' AND client.firstName="John" AND client.lastName="Smith"
GROUP BY orderLine.ProductID

Given a dataspace D, we define a query as q=($\mathbf{q_\pi}$ ,$\mathbf{q_\gamma}$ , $\mathbf{q_\sigma}$)

$q_\pi$ = {$f_6$}

$q_\gamma$ = {(f7,avg)}

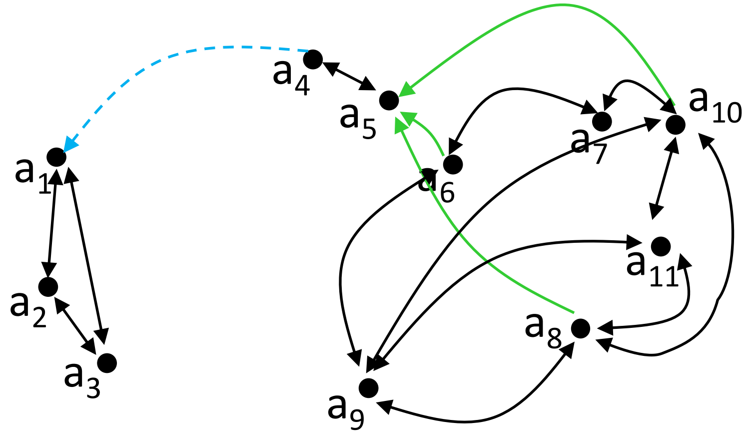
$q_\sigma$= {($f_2$,"John"), ($f_3$,"smith"), (f8,"ABC")}

# The Data Graph

To build a query plan we rely on a set of structural relationships between attributes that are modeled in a data graph

The **data graph** G is a graph G=(A*,E) where A* is the set of all the attributes of all databases while an edge $e \in E$ between two attributes a' and a'' indicates the existence of a relationship, which is described by its type *type(e)*

- **sibling**: represented as a' $\leftrightarrow$ a'', it indicates that a' and a'' are in the same collection and at the same nesting level;
- **nested**: represented as a' $\xrightarrow{n}$ a'', it indicates that a' is nested inside a'';
- **fk**: represented as a' a' $\xrightarrow{fk}$ a'', it indicates that the values of a'' are referred to the values of a'.




*More sibling arcs are omitted for readibility*

# The Query Graph

The **query graph** $G_q=(A' \subseteq A^*, E' \subseteq E)$ is the minimally connected subgraph of the data graph G such that:

1. $A' \supseteq attr(q)$ **all attributes involved in the query are in the query graph**
2. there exists $A'' \subseteq A'$, s.t. $A'' \neq \emptyset$, $A'' \supseteq q_\gamma$, $\forall (a \in A'', a' \in A')$ it is $a \Rightarrow a'$ **q can be answered since in the query are in the query graph since all the events are represented at the right level of granularity**

# Presentation Layout

Functional Architecture → Model Data & Data Structures → Build the Query Plan

# Build the Query Plan

**Algorithm 1** Definition of the NRA execution plan for a query $q$

**Input** $q = (q_\pi, q_\gamma, q_\sigma)$: a query; $G_q = (A', E')$ the query graph for $q$.
**Output** $P$: the NRA plan of $q$.

$P \leftarrow \varnothing$
$LP \leftarrow \varnothing$
$GP_q \leftarrow partitionQueryGraph(G_q)$
**for all** $GP_q^i \in GP_q$ **do**
  $CP \leftarrow \varnothing$
  $C \leftarrow identifyAccessedCollections(GP_q^i)$
  **for all** $col \in C$ **do**
    $CP_{col} \leftarrow defineCollectionPlan(col, GP_q^i)$
  $LP_i \leftarrow defineLocalJoins(CP, GP_q^i)$
$P \leftarrow defineGlobalPlan(LP, G_q)$
**return** $P$

**Split $G_q$: one split or each involved database & each query path**

exploit fk relationships & db( )

$a' \xrightarrow{fk} a''$ & $db(a') \neq db(a'')$

# Build the Query Plan

**Algorithm 1** Definition of the NRA execution plan for a query $q$

**Input** $q = (q_\pi, q_\gamma, q_\sigma)$: a query; $G_q = (A', E')$ the query graph for $q$.
**Output** $P$: the NRA plan of $q$.

$P \leftarrow \varnothing$
$LP \leftarrow \varnothing$
$GP_q \leftarrow partitionQueryGraph(G_q)$
**for all** $GP_q^i \in GP_q$ **do**
    $CP \leftarrow \varnothing$
    $C \leftarrow identifyAccessedCollections(GP_q^i)$
    **for all** $col \in C$ **do**
        $CP_{col} \leftarrow defineCollectionPlan(col, GP_q^i)$
    $LP_i \leftarrow defineLocalJoins(CP, GP_q^i)$
$P \leftarrow defineGlobalPlan(LP, G_q)$
**return** $P$

For each portion define the local plan

# Build the Query Plan

**Algorithm 1** Definition of the NRA execution plan for a query $q$

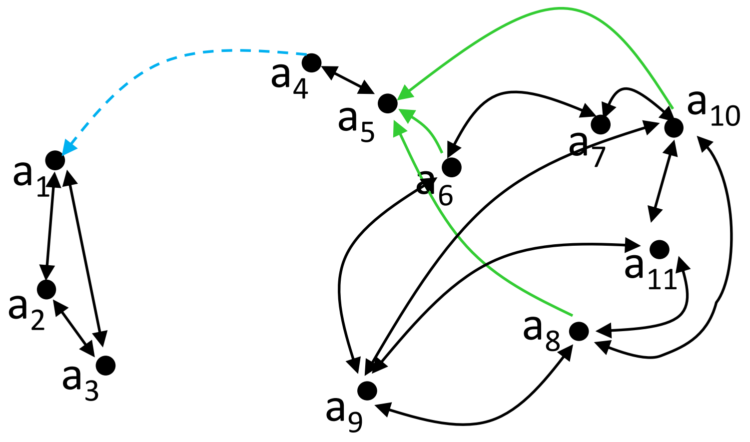
**Input** $q = (q_\pi, q_\gamma, q_\sigma)$: a query; $G_q = (A', E')$ the query graph for $q$.
**Output** $P$: the NRA plan of $q$.

$P \leftarrow \varnothing$
$LP \leftarrow \varnothing$
$GP_q \leftarrow partitionQueryGraph(G_q)$
**for all** $GP_q^i \in GP_q$ **do**
  $CP \leftarrow \varnothing$
  $C \leftarrow identifyAccessedCollections(GP_q^i)$
  **for all** $col \in C$ **do**
    $CP_{col} \leftarrow defineCollectionPlan(col, GP_q^i)$
  $LP_i \leftarrow defineLocalJoins(CP, GP_q^i)$
$P \leftarrow defineGlobalPlan(LP, G_q)$
**return** $P$

**Define the global plan:**

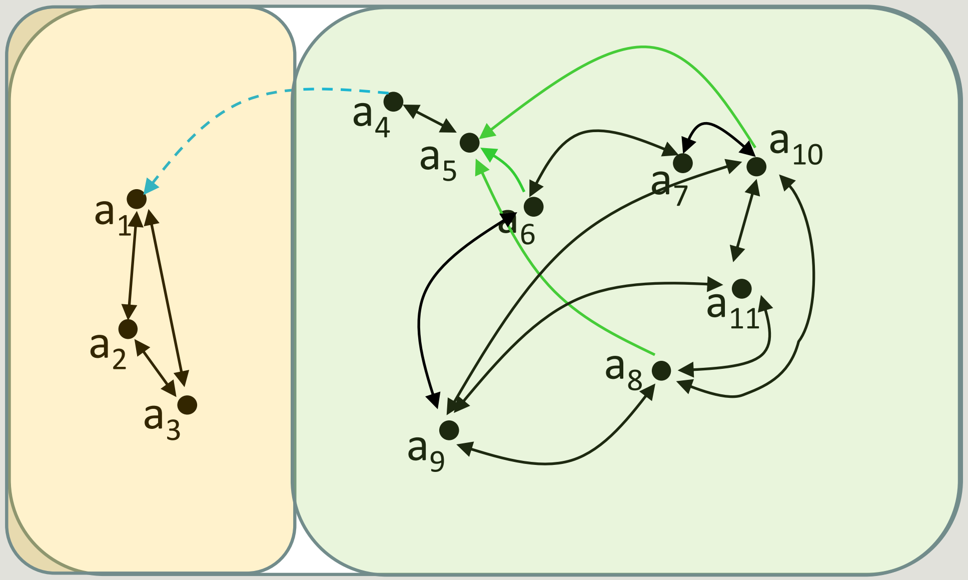
1. **Add a join condition for each split**
2. **Aggregate data to $q_\pi$ applying operators $q_\gamma$**

# Building the Plan: Split the Query Graph

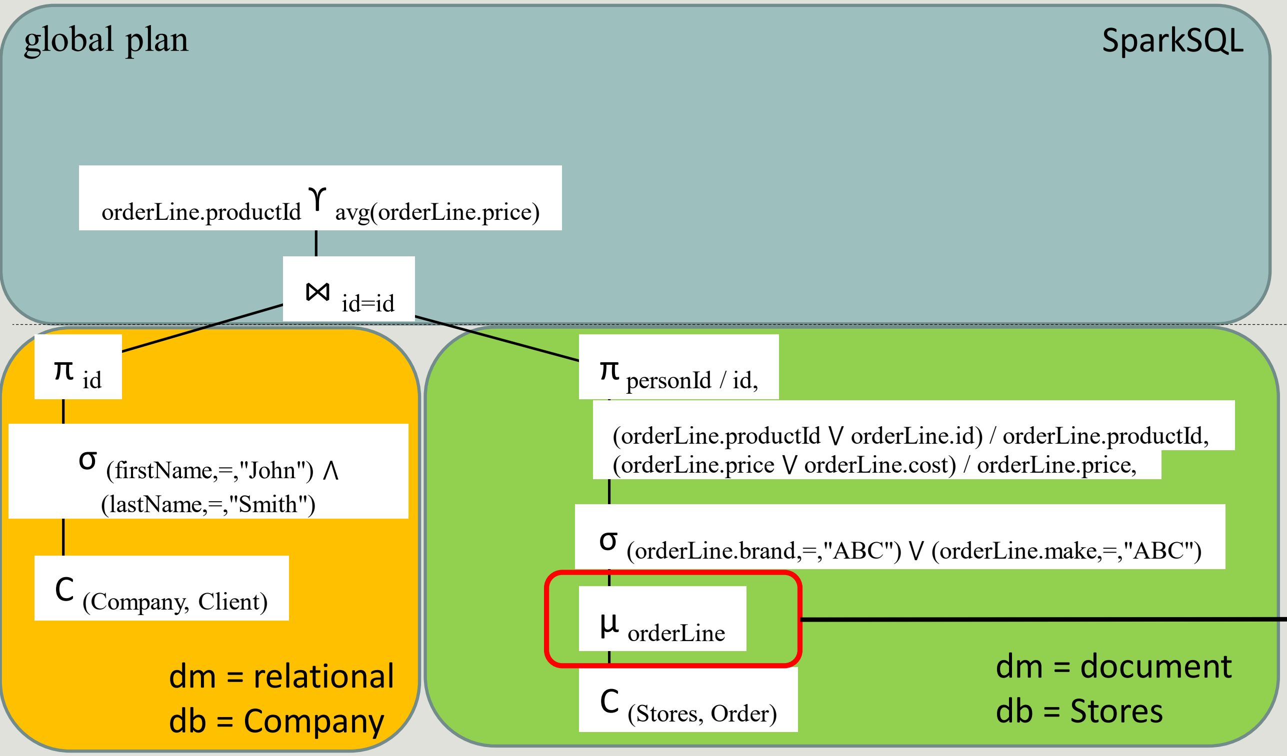


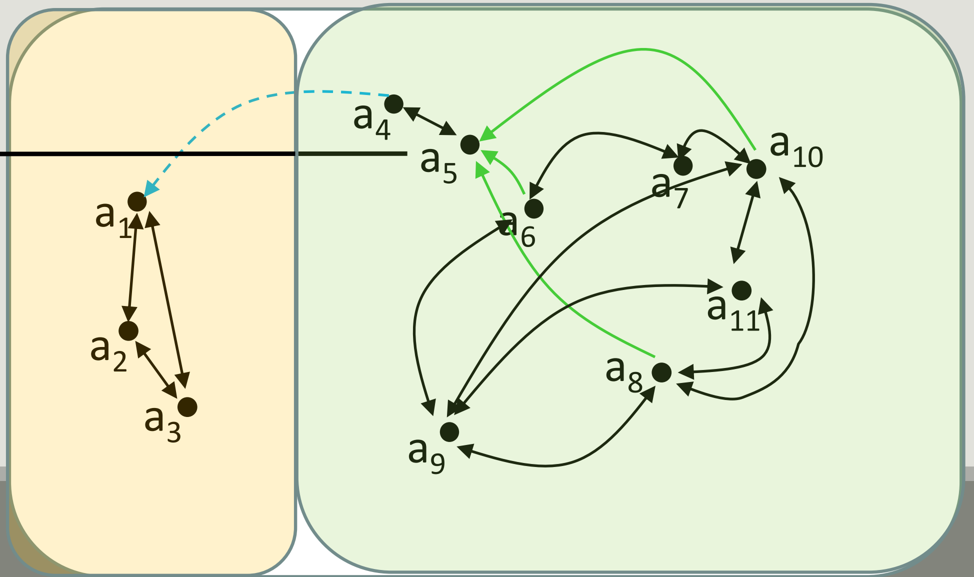

Query Graph

# Building the Plan: Split the Query Graph



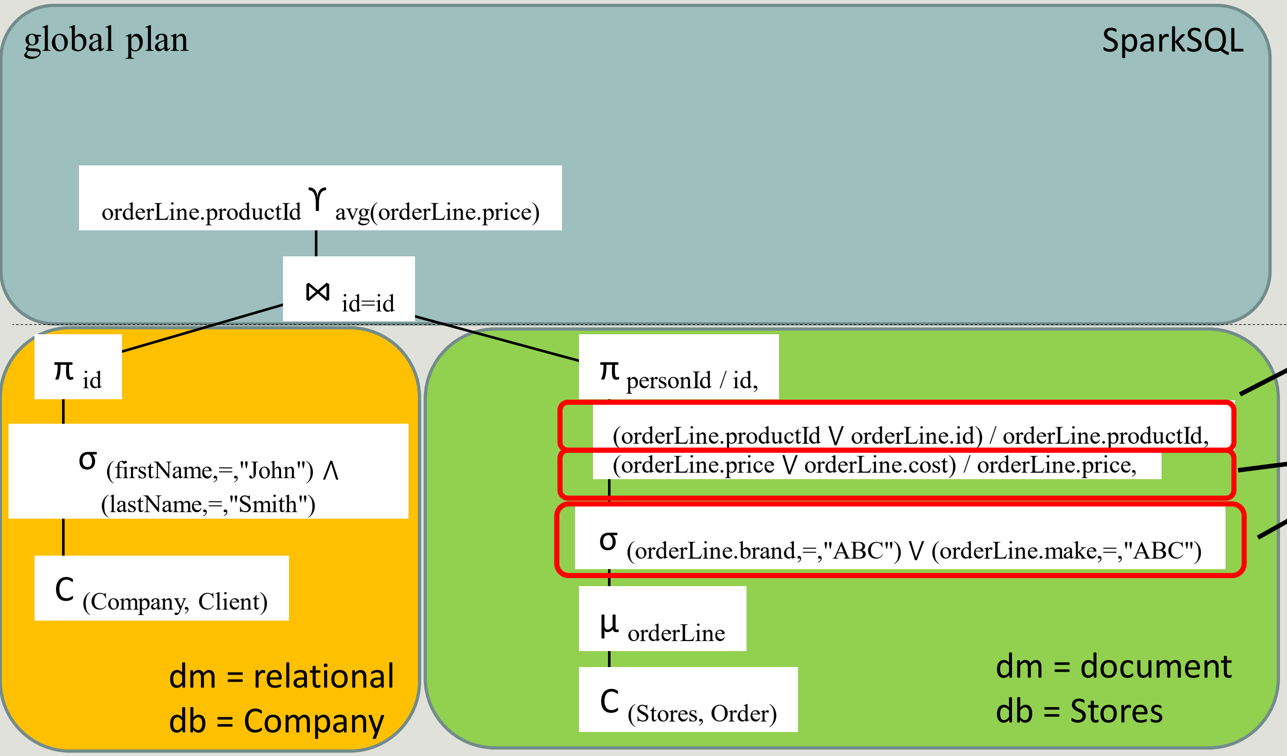
# Building the Plan: Define the Local Plans

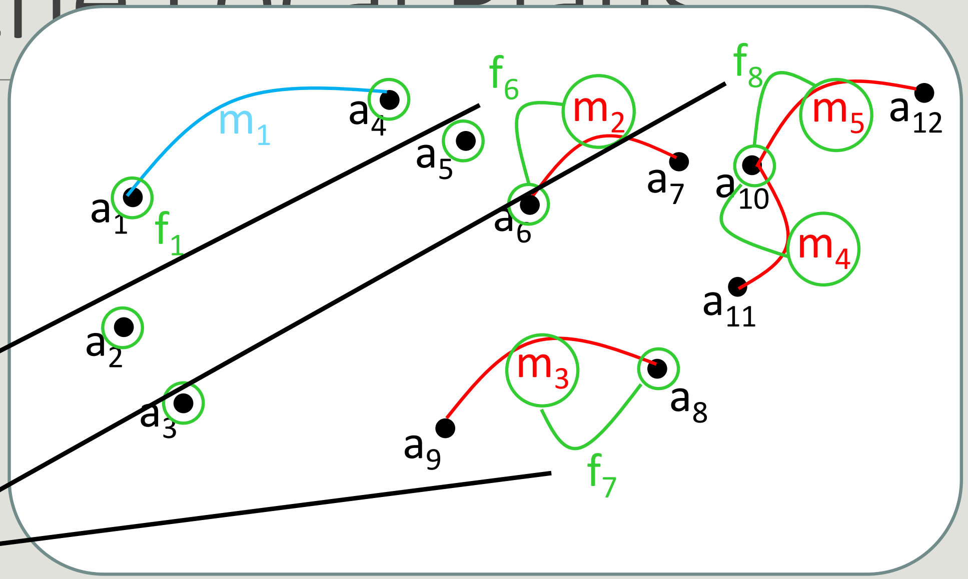


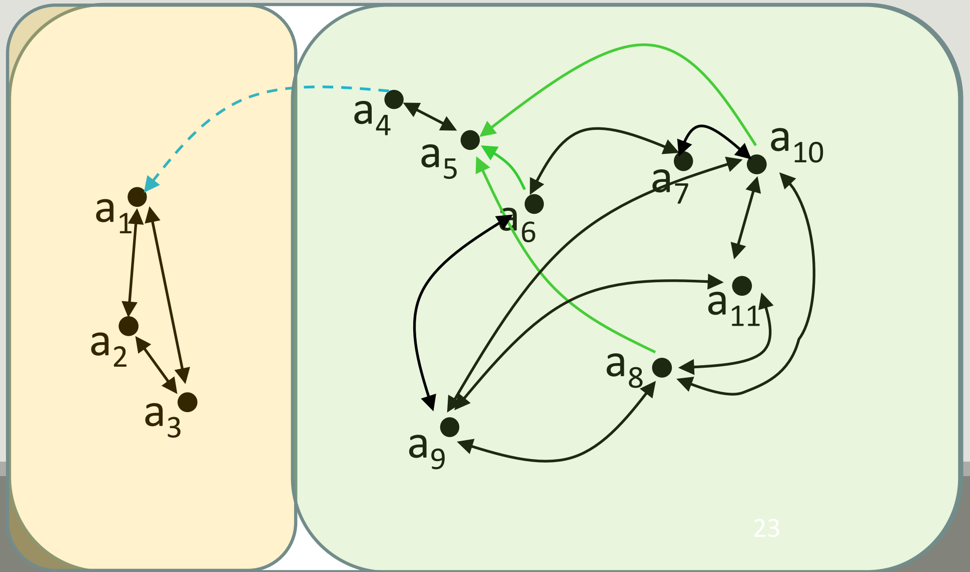




Nesting is solved
exploiting nested arcs

# Building the Plan: Define the Local Plans

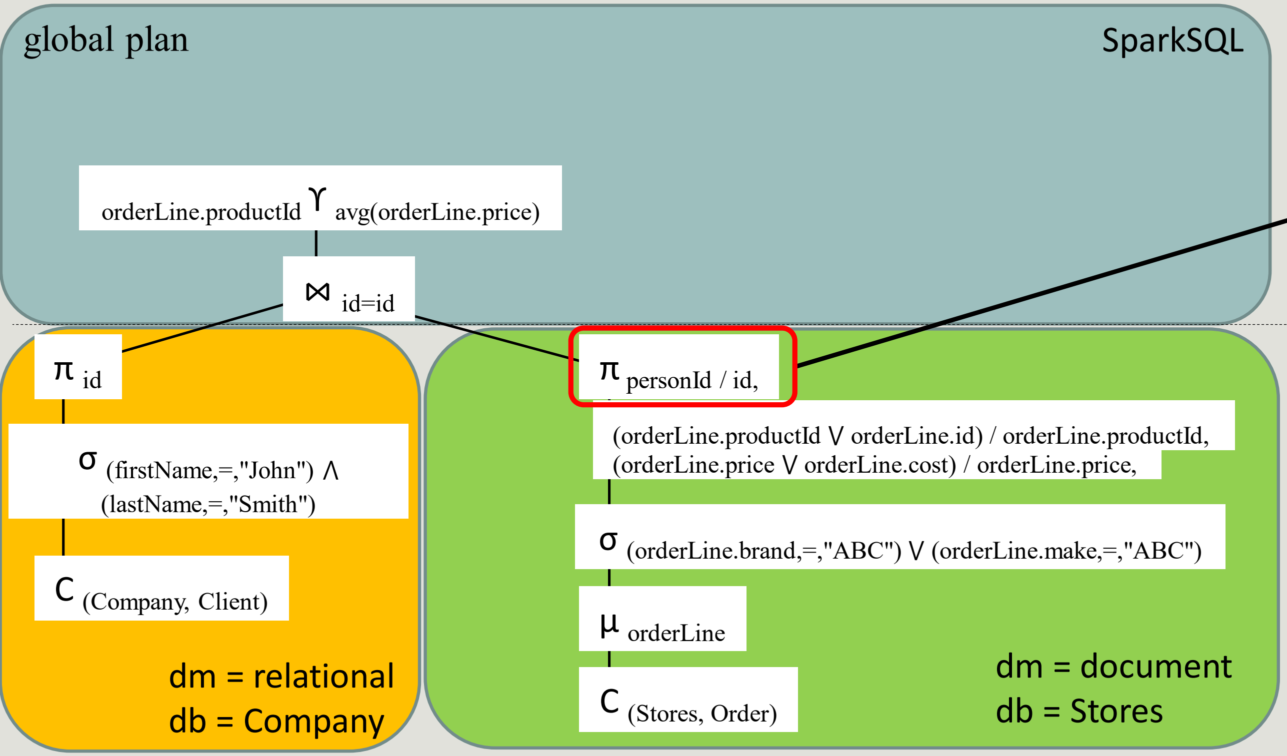


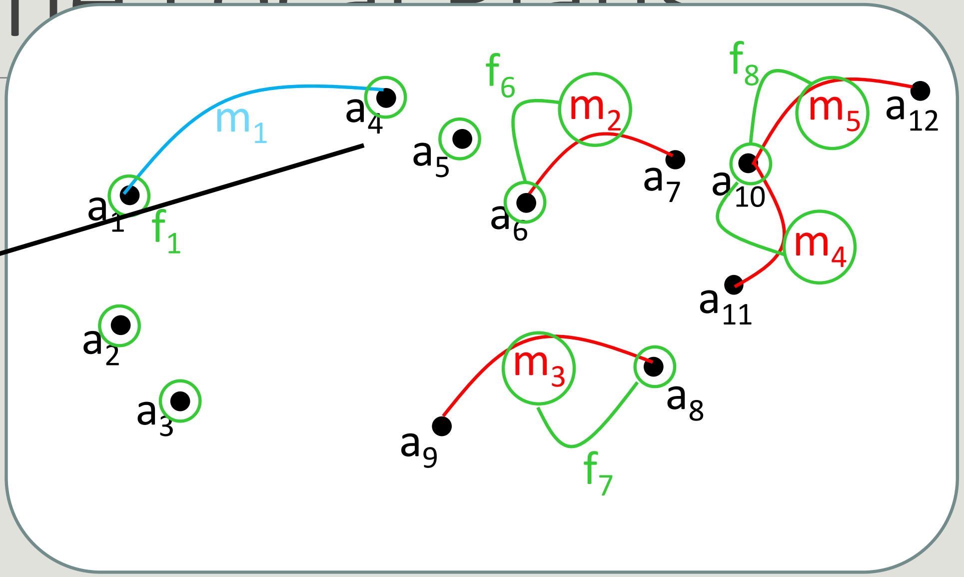



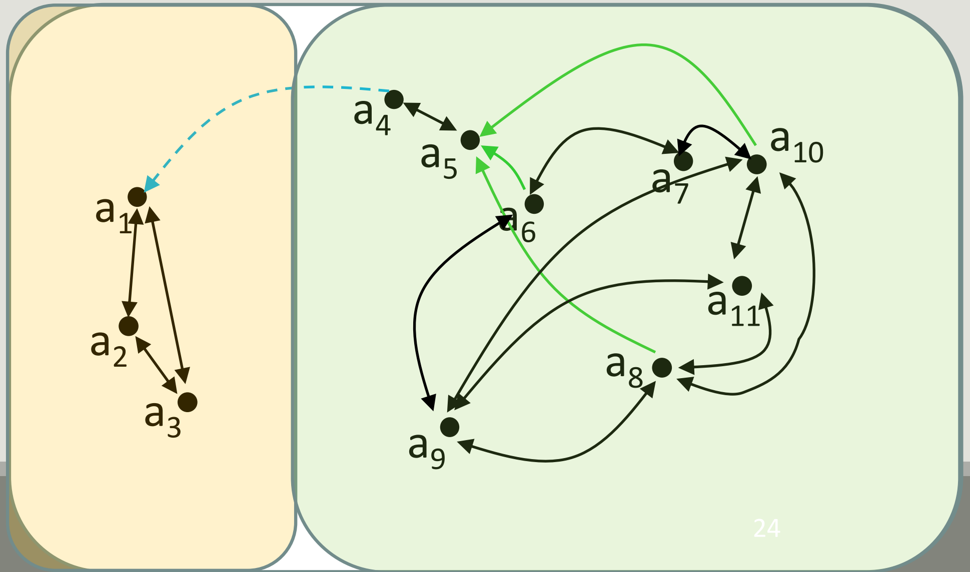




Features and mappings enable heterogenity handling

# Building the Plan: Define the Local Plans

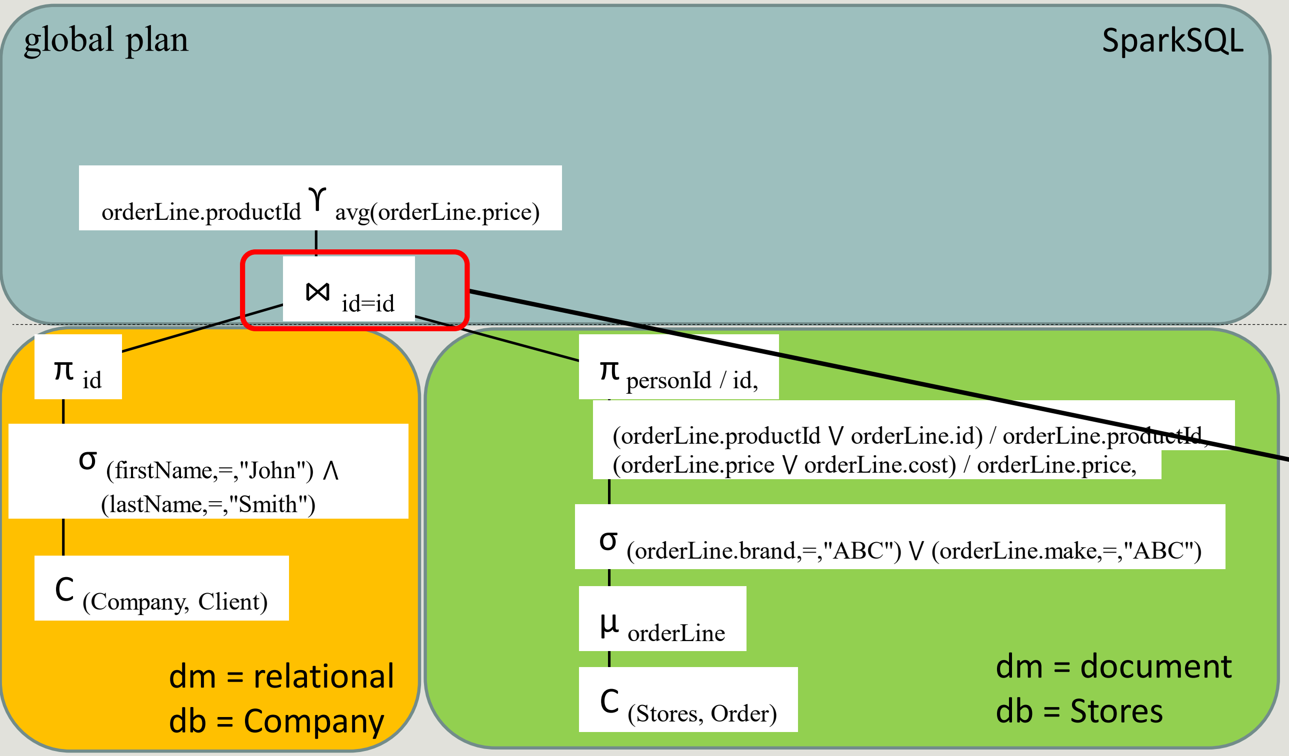


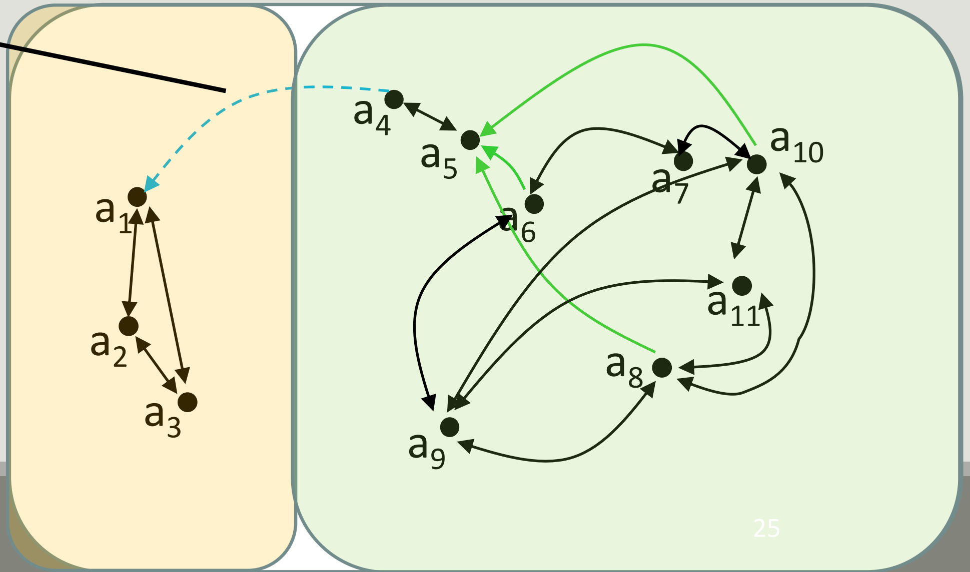




ojections solves semantic equivalence and data format heteroneity by applying transcoding functions φ

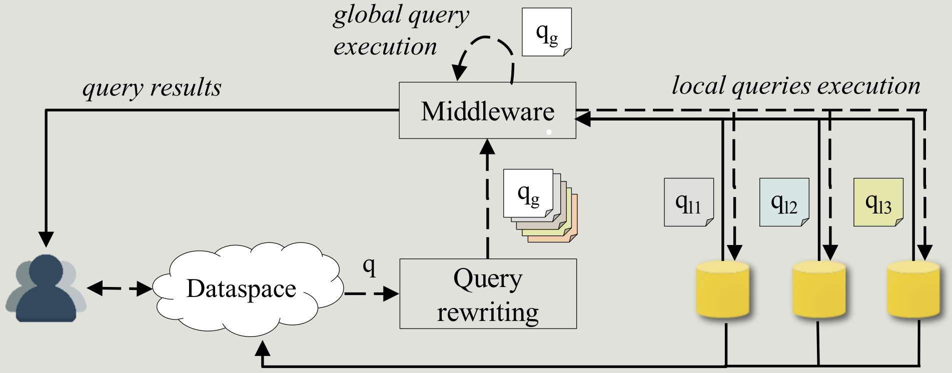
# Building the Plan: Define the Global Plan



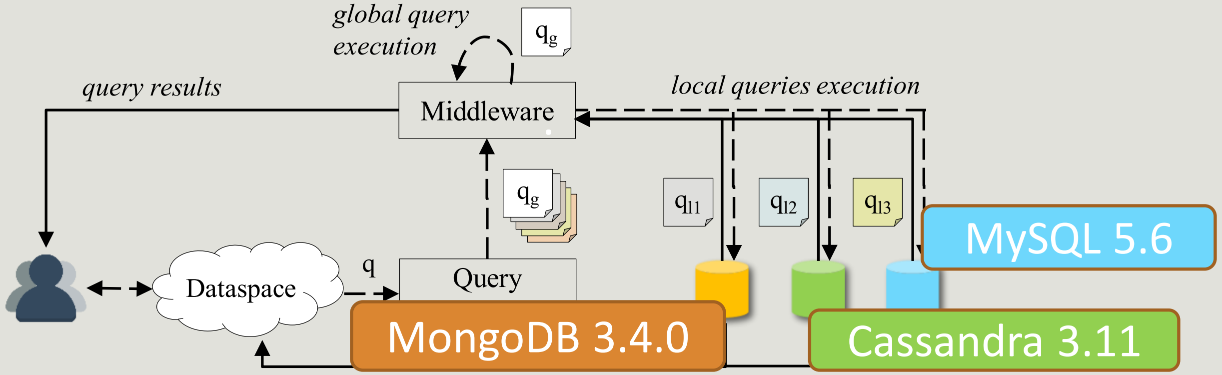

Partial results are joined: for each splits a join is added



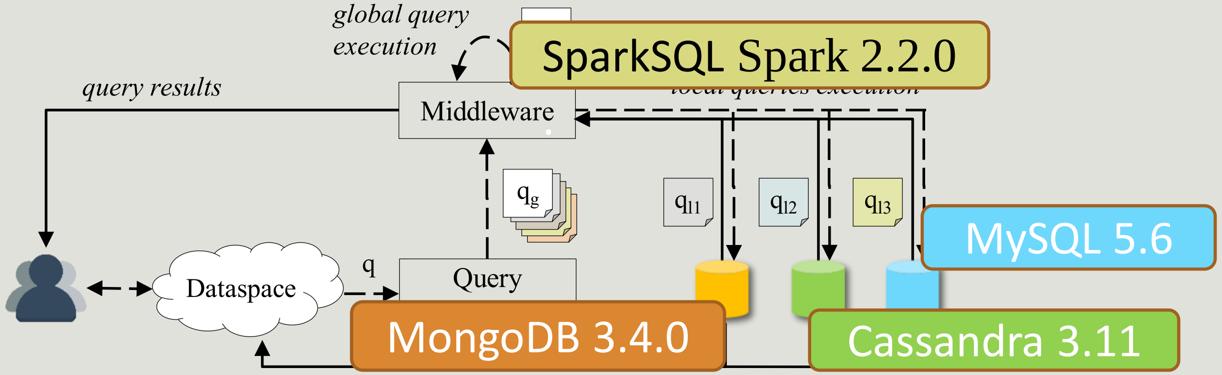
# Preliminary tests



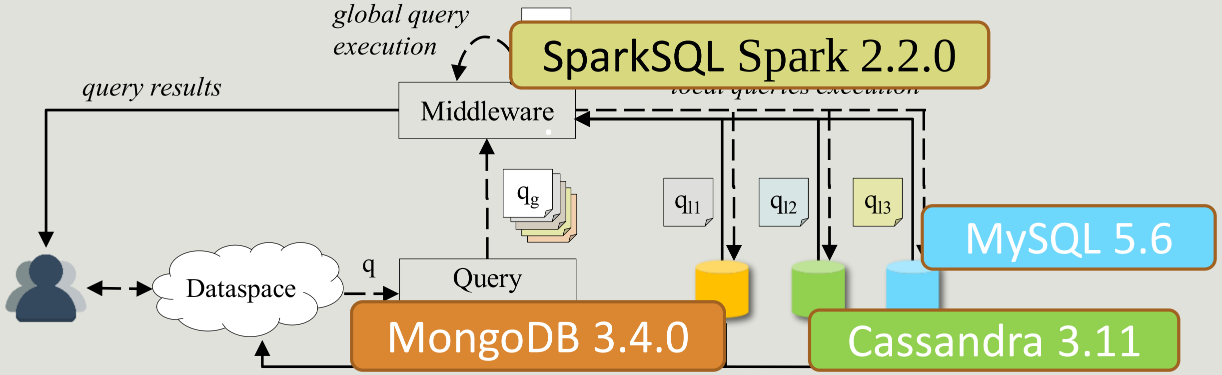
# Preliminary tests



# Preliminary tests

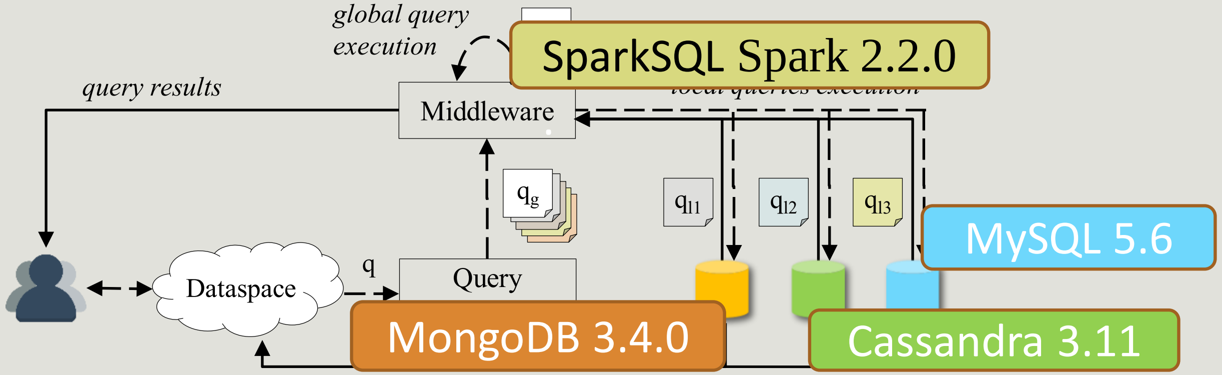


# Preliminary tests

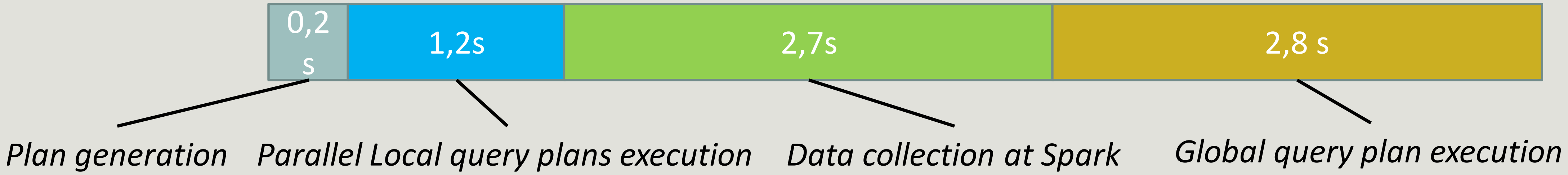



- Benchmark based on Unibench extended with heterogeneity, 142K records
- HW: single server Intel I5 i5-4670 - 3.4 GHz - 4 cores - 16 GB of RAM

# Preliminary tests




- Benchmark based on Unibench extended with heterogeneity, 142K records
- HW: single server Intel I5 i5-4670 - 3.4 GHz - 4 cores - 16 GB of RAM
- Execution of the running example query takes 6,9 secs



# Future Works

**Expressiveness**

- Cover horizontal partitioning of the data: the same information can span on several collections and on different DBs.
- Support additional data models (e.g., key-value and graph)
- Enable a broader set of queries than GPSJs

**User-system interaction**: introduce KPIs to provide further insights to the user with respect to the underlying heterogeneity of the data

**Complete prototype implementation**