CloudMdsQL: Querying Heterogeneous Cloud Data Stores

Patrick Valduriez

Inria, Montpellier, France

Joint work with Boyan Kolev (Inria), Ricardo Jimenez-Peris (U. Madrid), Norbert Martínez-Bazan (Sparcity), Jose Pereira (INESC)

Big Data Landscape

Easy to get lost

Many diverse solutions

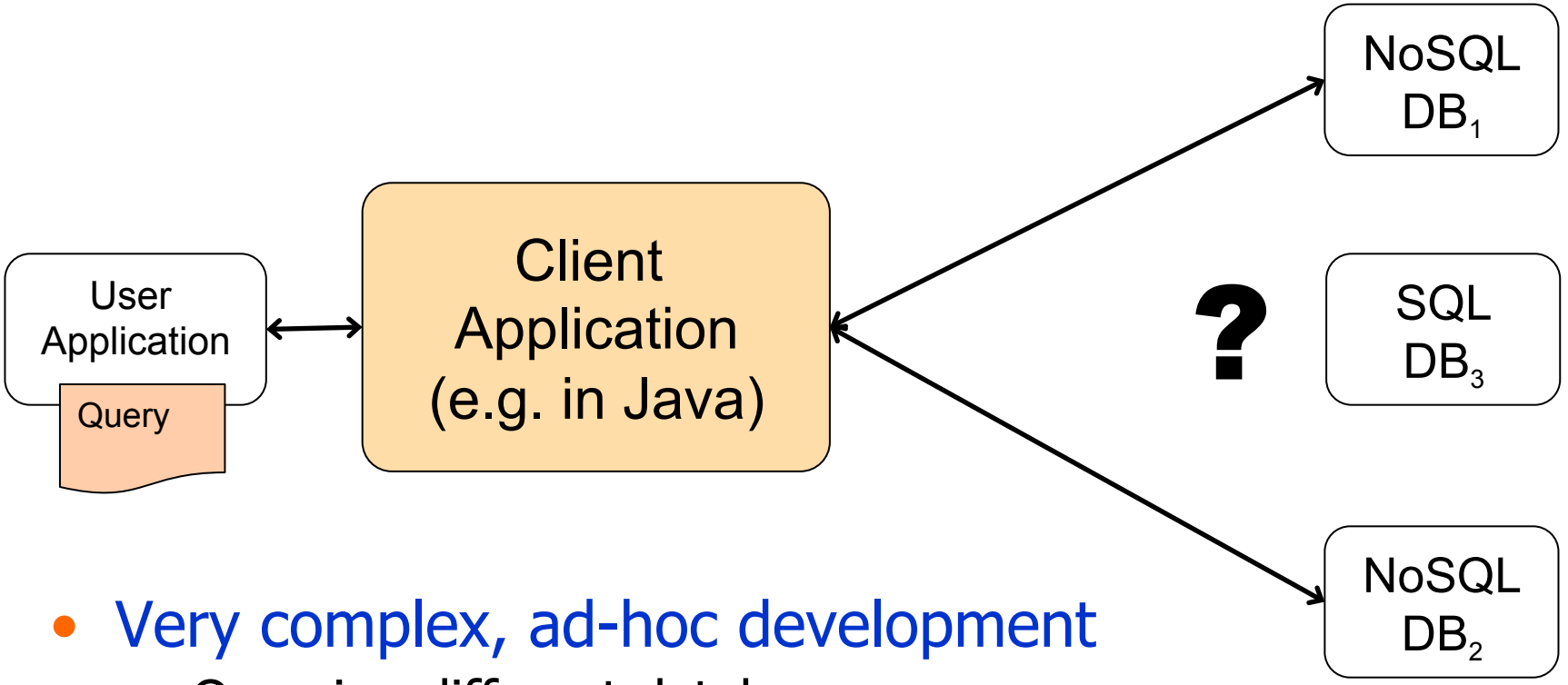
No standard

Keeps changing

dave@vcdave.com

blogs.forbes.com/davefeinleib

General Problem We Address



- Very complex, ad-hoc development
  - Querying different databases
  - Managing intermediate results
  - Delivering (e.g. sorting) the final results
- Hard to extend
  - What if a new SQL DB appears?

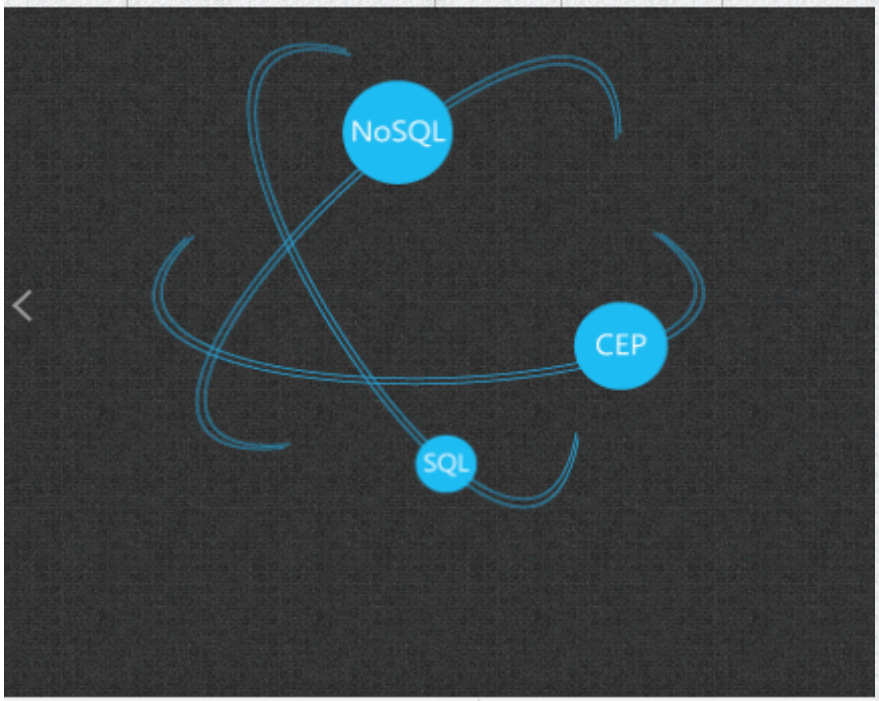
Outline

- The CoherentPaaS IP project
- Related work and background
- CloudMdsQL objectives
- Design decisions
- Data model
- Query language
- Query rewriting
- Validation

FP7 IP project
(2013-2016, 6 M€)

Coh

Home | About CoherentPaaS | News | Partners | Case studi



Coherence

Transactional semantics across cloud data stores

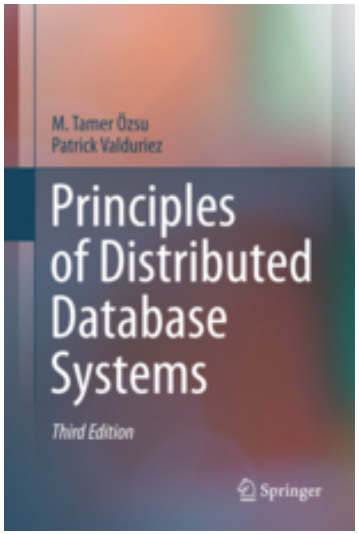
Scalability

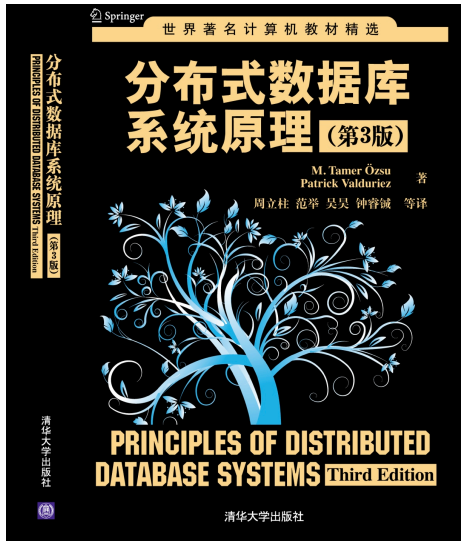
Ultra-scalable preserving ACID properties

Universidad Politecnica de Madrid (Coordinator)	UPM	Spain
Neurocom SA	Neurocom	Greece
INRIA	INRIA	France
Foundation for Research and Technology – Hellas	FORTH	Greece
Institute of Engineering Systems and Computers	INESC	Portugal
Sparsity	Sparsity	Spain
MonetDB	MonetDB	Netherlands
QuartetFS	QuartetFS	United Kingdom
Institute of Communication and Computer Systems	ICCS	Greece
Portugal Telecom Innovaçao	PTIN	Portugal

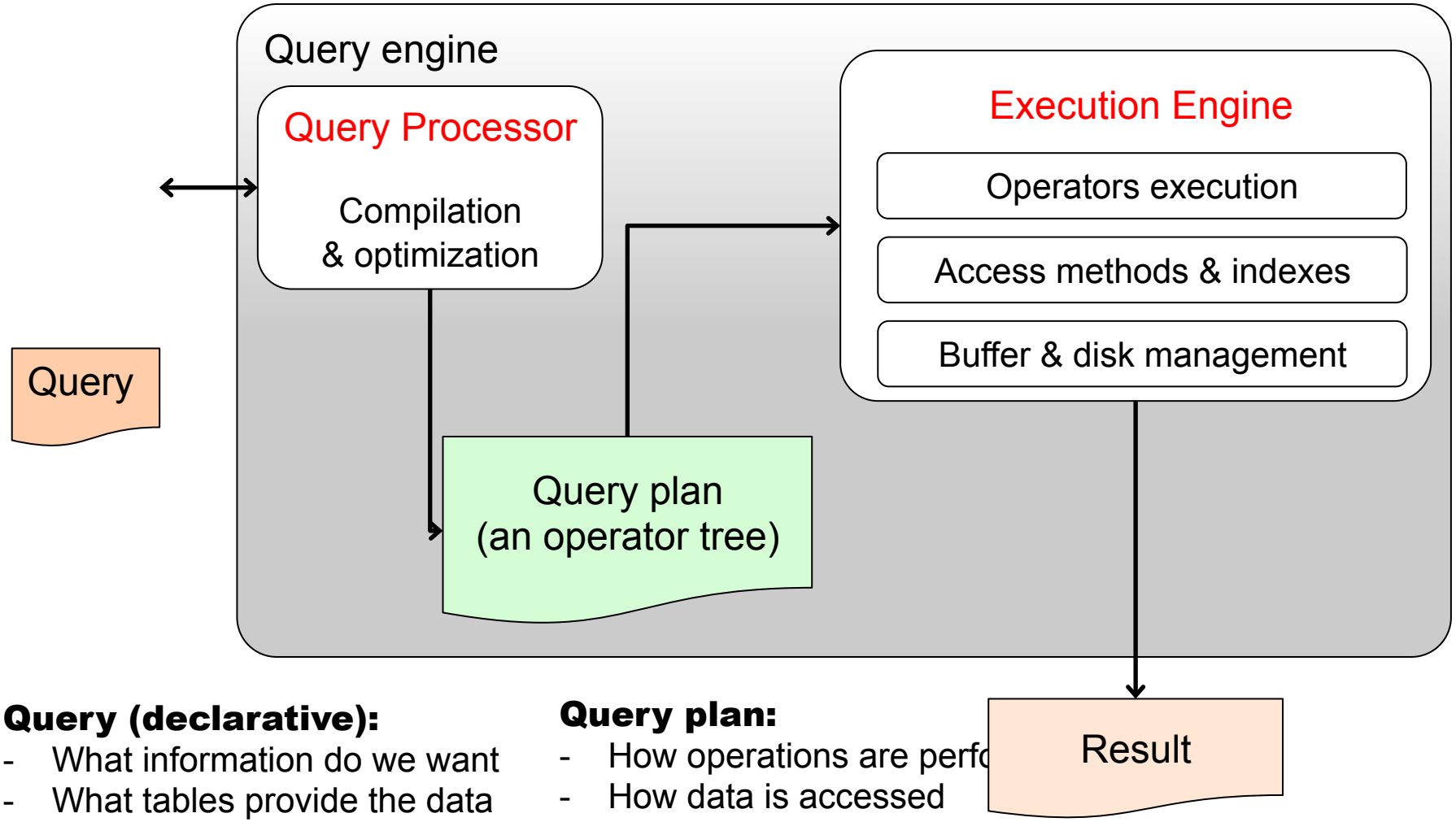
Related Work

- Multidatabase systems (or federated database systems)
  - A few databases (e.g. less than 10)
    - Corporate DBs
  - Powerful queries (with updates and transactions)
- Web data integration systems
  - Many data sources (e.g. 1000's)
    - DBs or files behind a web server
  - Simple queries (read-only)
- Mediator/wrapper architecture





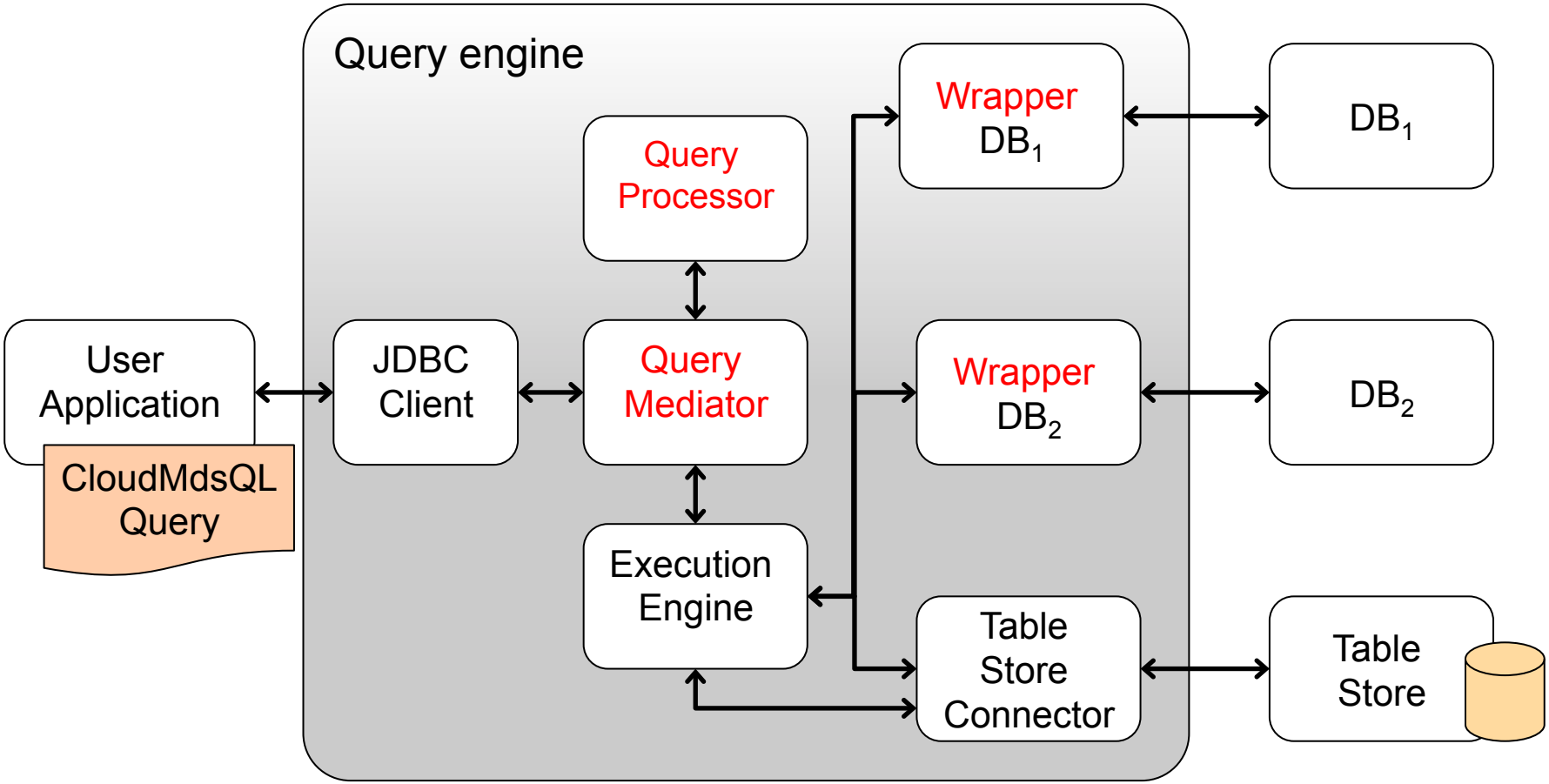
Background: DB query processing



Query (declarative):
- What information do we want
- What tables provide the data

Query plan:
- How operations are perfo
- How data is accessed

CoherentPaaS Query Engine



CloudMdsQL Objectives

- Design an SQL-like query language to query multiple databases (SQL, NoSQL) in a cloud
  - Autonomous databases
    - This is different from recent multistore systems (no autonomy)
- Design a query engine for that language
  - Query processor
    - To produce an efficient execution plan
  - Execution engine
    - To run the query, by calling the data stores and integrating the results
- Validate with a prototype
  - With multiple data stores: MonetDB, Sparksee, MongoDB, Derby, Hbase, etc.

Issues

- No standard in NoSQL
  - Many different systems
    - Key-value store, big table store, document DBs, graph DBs
- Designing a new language is hard and takes time
  - We should not reinvent the wheel
  - Start simple and useful
- We need to set precise requirements
  - In increasing order of functionality
    - Start simple and useful (and efficient)
  - Guided by the CoherentPaaS project uses cases
    - E.g. bibliography search

Requirements for MDB Query Languages*

1. Nested queries
   - Allow queries to be arbitrarily chained together in sequences, so the result of one query (for one DB) may be used as the input of another (for another DB)
2. Data-metadata transformation
   - To deal with heterogeneous formats by transforming data into metadata and conversely
     - e.g. data into attribute or relation names, attribute names into relation names, relation names into data
3. Schema independence
   - Allows the user to formulate queries that are robust in front of schema evolution

* C. M. Wyss, E.L. Robertson. Relational Languages for Metadata Integration. ACM TODS, 2005.

Design Considerations for CloudMdsQL

- Not for web data integration!
  - A query is for a few DBs
    - And needs to have access rights to each DB
- The DBs may have very different languages
  - No single language can capture all the others
    - E.g. SQL cannot express path traversal
- NoSQL DBs can be schemaless
  - Makes it (almost) impossible to derive a global schema
- We need to express powerful queries
  - To exploit the full power of the different DB languages
    - E.g. perform a path traversal in a graph DB

Our Design Choices

- Data model: schemaless, table-based
  - With rich data types
    - To allow computing on typed values
  - No global schema and schema mappings to define
- Query language: functional-style SQL*
  - SQL widely accepted
  - Can represent all query building blocks as functions
    - A function can be expressed in one of the DB languages
  - Function results can be used as input to subsequent functions
    - Supports requirement (1) of MDB languages
  - Functions can transform types and do data-metadata conversion
    - Supports requirement (2) of MDB languages

*P. Valduriez, S. Danforth. Functional SQL, an SQL Upward Compatible Database Programming Language. Information Sciences, 1992.

*C. Binnig et al. FunSQL: it is time to make SQL functional. EDBT/ICDT, 2012.

CloudMdsQL Data Model

- A kind of nested relational model
  - With JSON flavor
- Data types
  - Basic types: int, float, string, id, idref, timestamp, url, xml, etc. with associated functions (+, concat, etc.)
  - Type constructors
    - Row (called *object* in JSON): an unordered collection of (attribute : value) pairs, denoted by { }
    - Array: a sequence of values, denoted by [ ]
- Set-oriented
  - *A table* is a named collection of rows, denoted by Table-name ()

Data Model – examples*

**Any resemblance to living persons is coincidental*

- Key-value

```
Scientists ({key:"Ricardo", value:"UPM, Spain"},
           {key:"Martin", value:"CWI, Netherlands"})
```

- Relational

```
Scientists ({name:"Ricardo", affiliation:"UPM", country:"Spain"},
           {name:"Martin", affiliation:"CWI", country:"Netherlands"})
Pubs ({id:1, title:"Snapshot isolation", Author:"Ricardo", Year:2005})
```

- Document

```
Reviews ({PID: “1”, reviewer: “Martin”, date: “2012-11-18”,
 tags : ["implementation", "performance"],
 comments :
 [ { when : Date("2012-09-19"), comment : "I like it." },
   {when : Date("2012-09-20"), comment : "I agree with you." } ] })
```

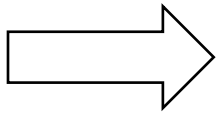
Query Language Requirements

- Define *named* table expressions
- Invoke specific API methods to query NoSQL data stores
- Convert arbitrary datasets to tables in order to comply with the common data model
- Complement the query language with functional capabilities
  - Perform data-metadata transformations
  - Perform type conversions

Python as the Functional Extension

- Why Python?
  - Supports all data types from the common data model
  - Many DBMSs have Python APIs (including Sparksee, MongoDB and MonetDB)
  - Simple, well-known and rich in standard libraries
- Example of Python expression that produces a table

```
T3(x int, a string) = {*
  for i in range(0,3):
    yield ( i+1, 'a'*(i+1) )
*}

SELECT x, a
FROM T3
```



x	a
1	a
2	aa
3	aaa

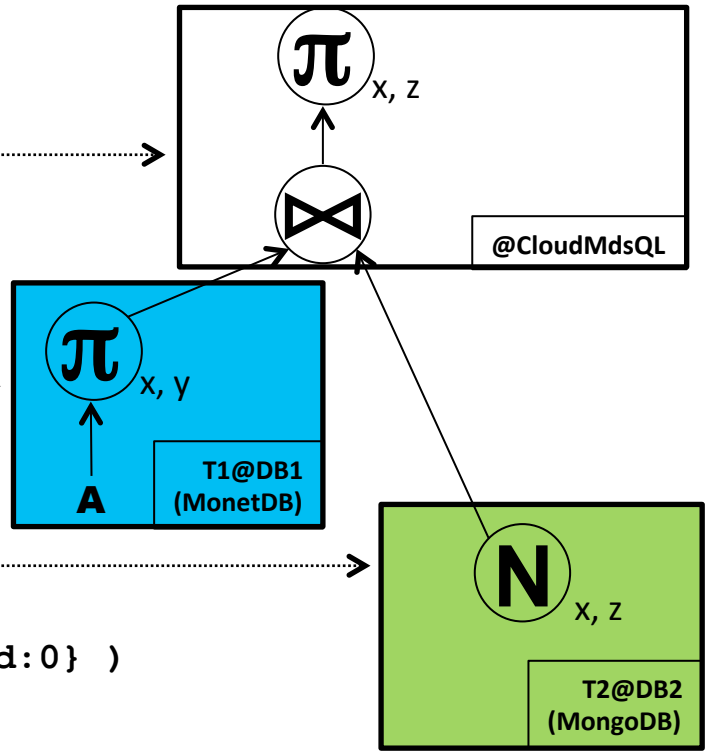
Table Expressions

- Named table expression
  - Expression that returns a table representing a nested query [against a data store]
  - Name and Signature (names and types of attributes)
  - Query is executed in the context of an ad-hoc schema
- 3 kinds of table expressions
  - Native named tables
    - Using a data store's native query mechanism
  - SQL named tables
    - Regular SELECT statements, for SQL-friendly data stores
  - Python named tables
    - Embedded blocks of Python statements that produce tables

CloudMdsQL Example

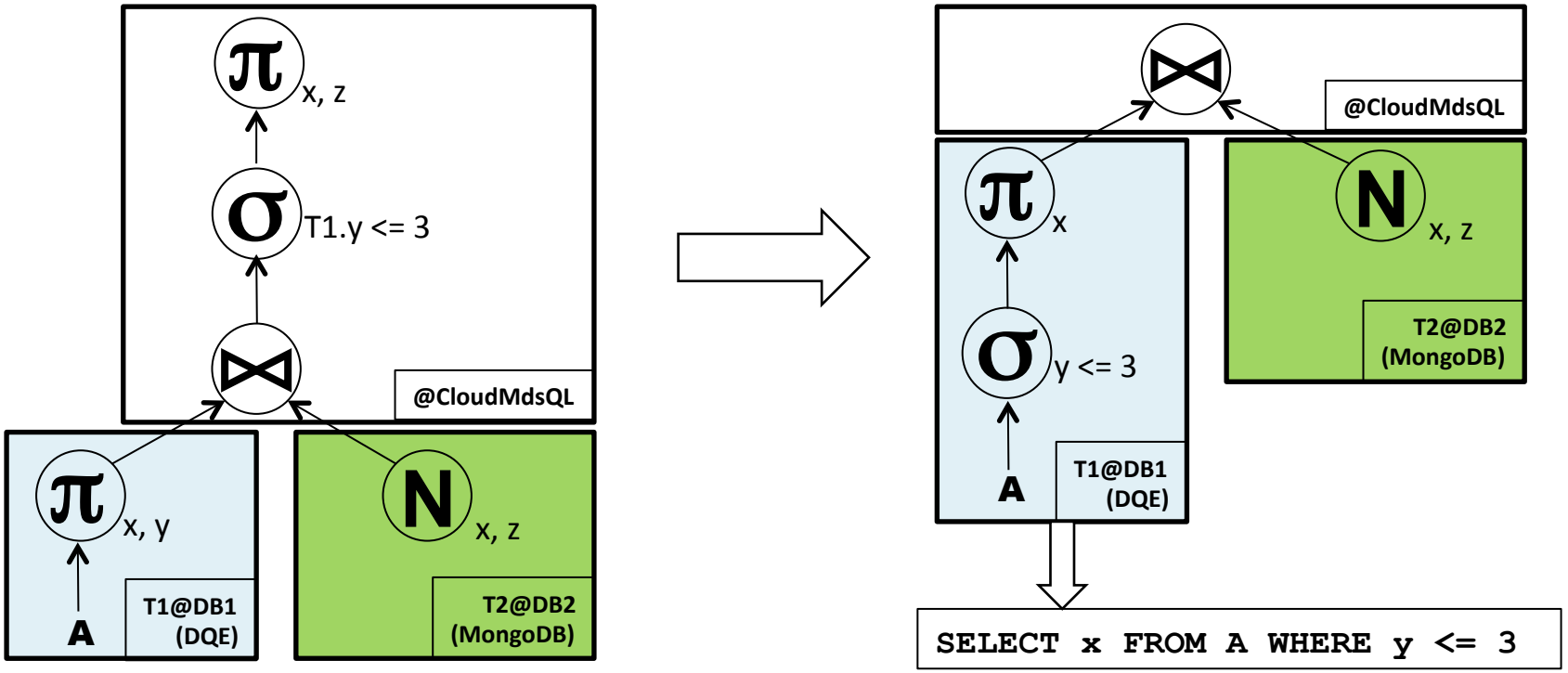
- A query that integrates data from:
  - DB1 – relational (MonetDB)
  - DB2 – document (MongoDB)

```
/* Integration query */
SELECT T1.x, T2.z
FROM T1 JOIN T2
  ON T1.x = T2.x

/* SQL sub-query */
T1(x int, y int)@DB1 =
( SELECT x, y FROM A )

/* Native sub-query */
T2(x int, z string)@DB2 = {*
  db.B.find( {$lt: {x, 10}}, {x:1, z:1, _id:0} )
*}
```

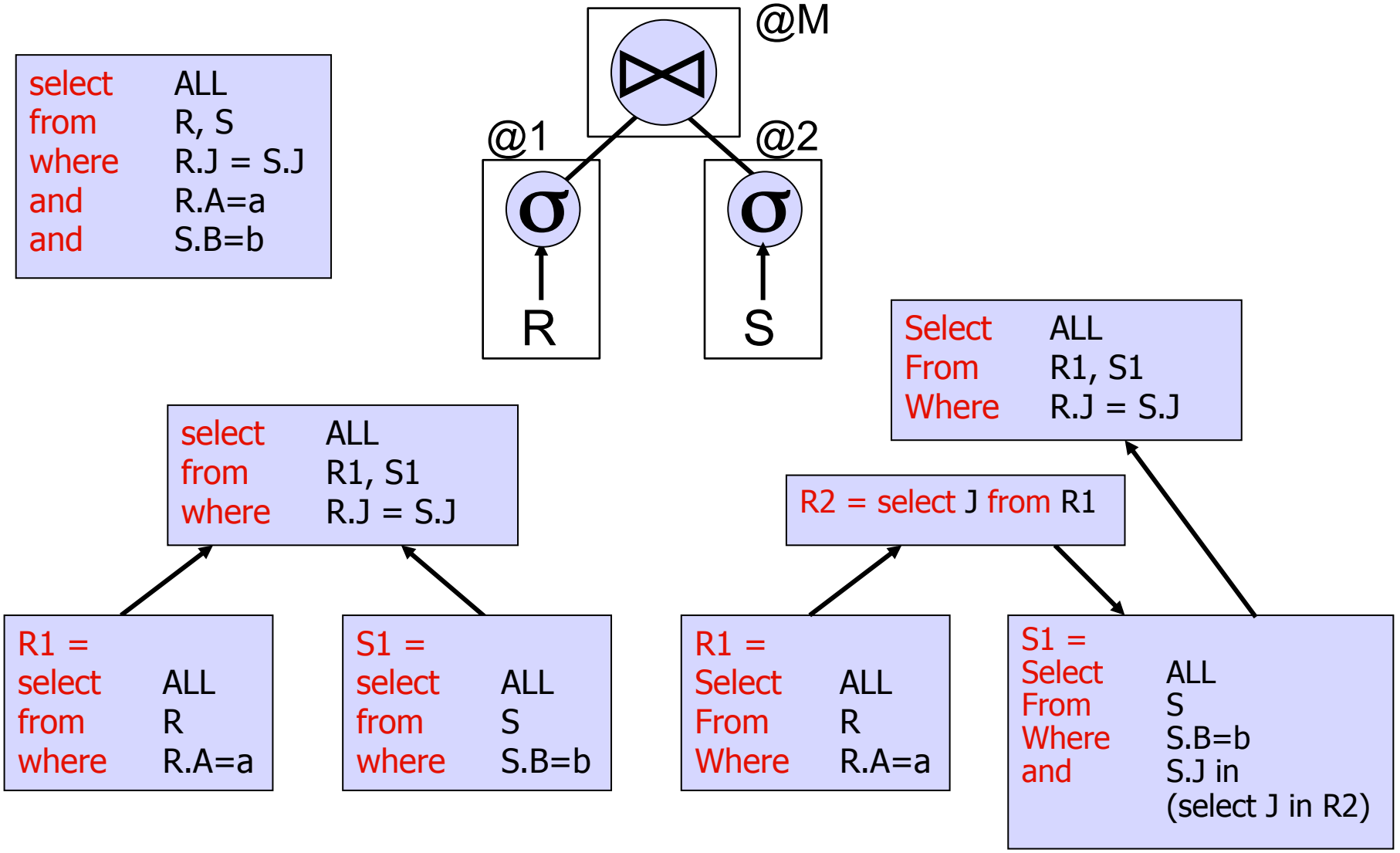


Sub-query Rewriting: selection pushdown

```
T1(x int, y int)@DB1 = ( SELECT x, y FROM A )

T2(x int, z string)@DB2 = {*
  db.B.find( {$lt: {x, 10}}, {x:1, z:1, _id:0} )
*}

SELECT T1.x, T2.z
FROM T1, T2
WHERE T1.x = T2.x AND T1.y <= 3
```



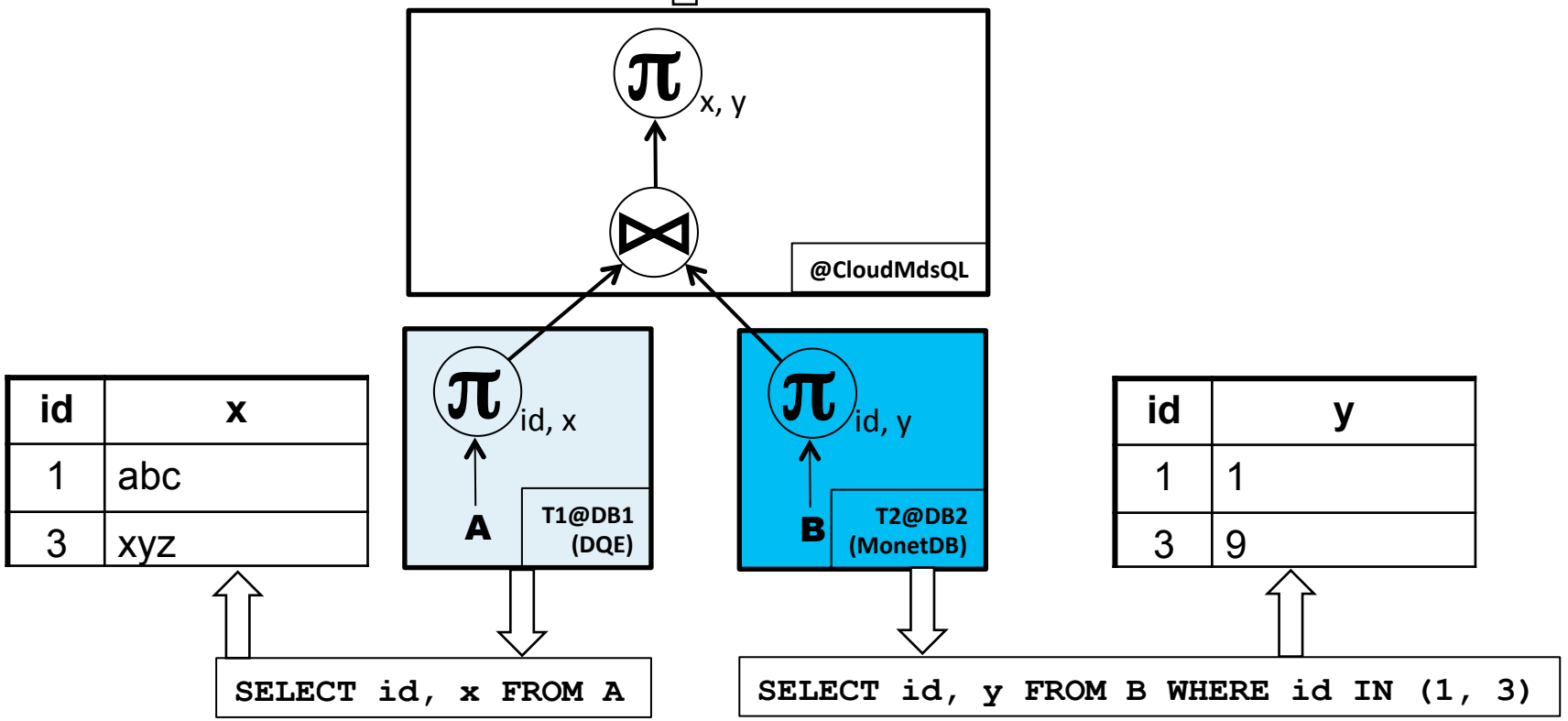
Optimization with Bindjoin



Sub-query Rewriting: bindjoin

```
T1(id int, x str...)@DB1 (SELECT id, x)
T2(id int, y int...)@DB2 (SELECT id, y FROM R2 )
SELECT T1.x, T2.y
FROM T1 BIND JOIN T2 ON T1.id = T2.id
```

x	y
abc	1
xyz	9



Validation

- Set up
  - Compiler/optimizer implemented in C++ (using the Boost.Spirit framework)
  - Operator engine (C++) based on the query operators of the Sparksee query engine
  - Query processor (Java) interacts with the above two components through the Java Native Interface (JNI)
  - The wrappers are Java classes implementing a common interface used by the query processor to interact with them
- 3 data stores
  - Relational: MonetDB
  - Document: MongoDB
  - Graph: Sparksee

Example DBs

DB1: a relational DB

Table Scientists (Name char(20), Affiliation char(10), Country char(30)
Table Pubs (ID int, Title char(50), Author char(20), Date date)

Scientists

Name	Affiliation	Country
Ricardo	UPM	Spain
Martin	CWI	Netherlands
Patrick	INRIA	France
Boyan	INRIA	France
Larri	UPC	Spain
Rui	INESC	Portugal

Pubs

ID	Title	Author	Date
1	Snapshot isolation in …	Ricardo	2012.11.10
5	Principles of DDBS	Patrick	2011.02.18
9	Graph DBs	Larri	2013.01.06

Example DBs (cont.)

DB2: a document DB (with SQL interface)

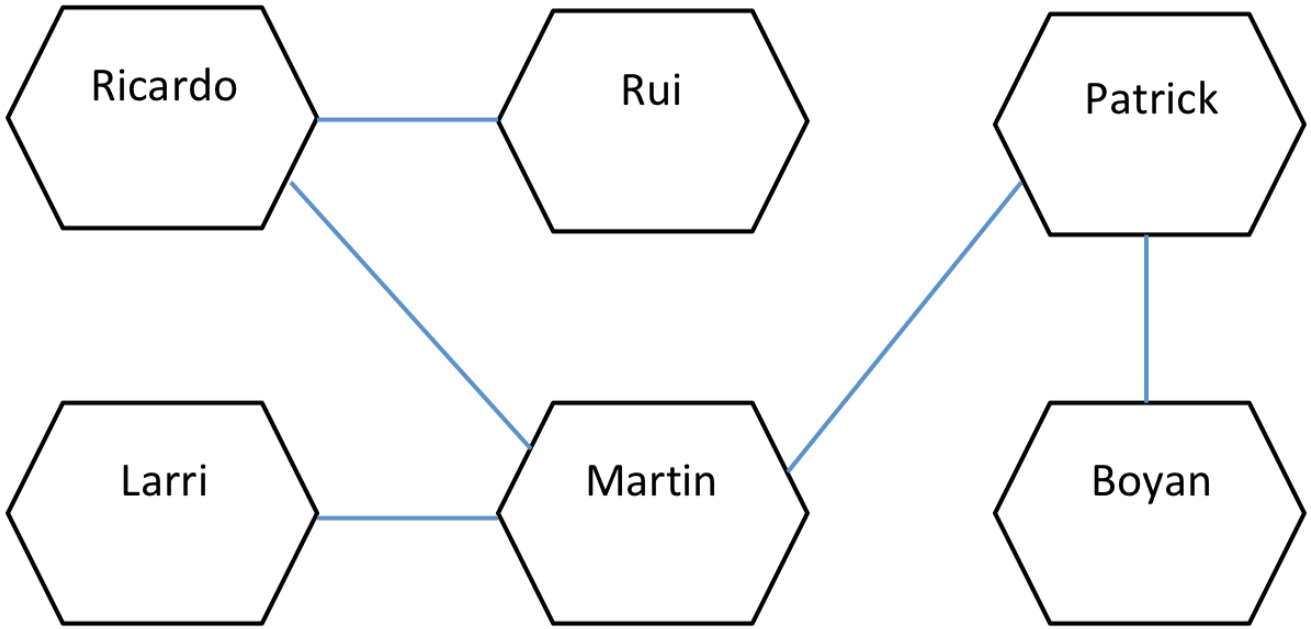
Reviews (PID string, reviewer string, date string, review string)

Reviews (
{PID: “1”, reviewer: “Martin”, date: “2012.11.18”, review: “… text …”},
{PID: “5”, reviewer: “Rui”, date: “2013.02.28”, review: “… text …”},
{PID: “5”, reviewer: “Ricardo”, date: “2013.02.24”, review: “… text …”},
{PID: “9”, reviewer: “Patrick”, date: “2013.01.19”, review: “… text …”})

Example DBs (cont.)

DB3: a graph DB

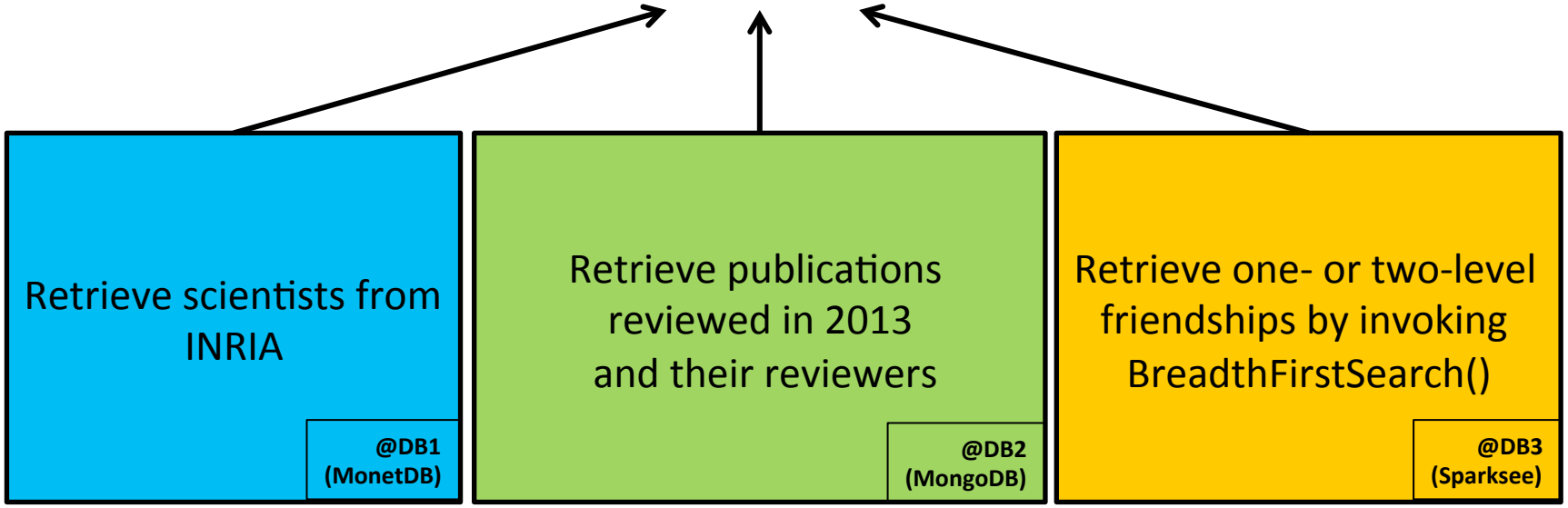
Person (name string, …) is_friend_of Person (name string, …)



CloudMdsQL Query: goal

Find conflicts of interest for papers from INRIA reviewed in 2013

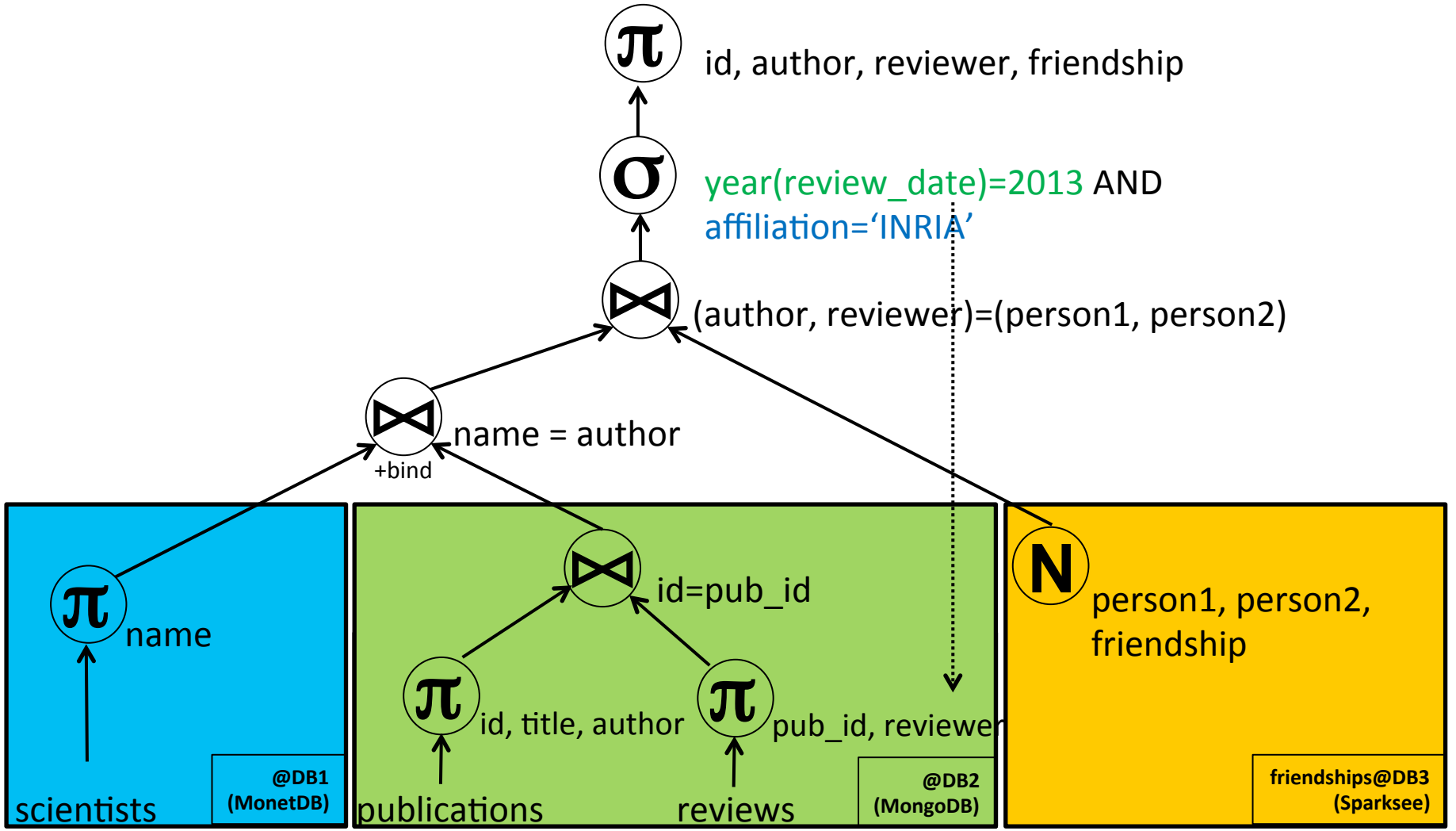
Retrieve papers by scientists from INRIA
that are reviewed in 2013
where the reviewer is a friend or friend-of-friend of the author



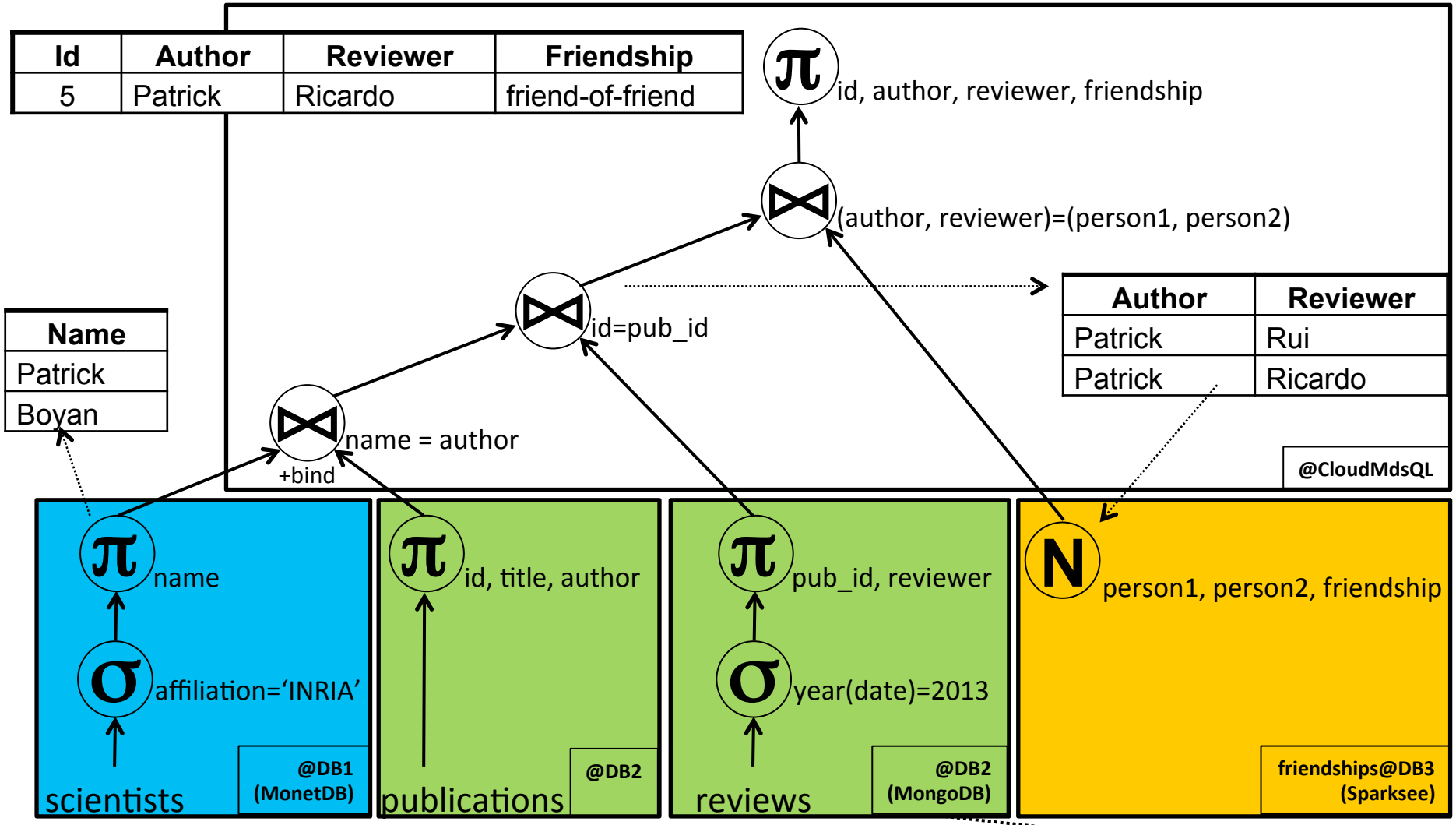
CloudMdsQL Query: expression

```
scientists( name string, aff string )@DB1 = (
  SELECT name, affiliation FROM scientists
)

pubs_revs( p_id, title, author, reviewer, review_date )@DB2 = (
  SELECT p.id, p.title, p.author, r.reviewer, r.date
  FROM publications p, reviews r
  WHERE p.id = r.pub_id
)

friendships( person1 string, person2 string, friendship string
             JOINED ON person1, person2 )@DB3 =
{*
  for (p1, p2) in CloudMdsQL.Outer:
    sp = graph.FindShortestPathByName( p1, p2, max_hops=2)
    if sp.exists():
      yield (p1, p2, 'friend' + '-of-friend' * sp.get_cost())
*}

SELECT pr.id, pr.author, pr.reviewer, f.friendship
FROM scientists s
  BIND JOIN pubs_revs pr ON s.name = pr.author
  JOIN friendships f ON pr.author = f.person1 AND pr.reviewer = f.person2
WHERE pr.review_date BETWEEN '2013-01-01' AND '2013-12-31' AND s.aff = 'INRIA';
```

Initial Query Plan



Rewritten Query Plan



CloudMdsQL Contributions

- Advantage
  - Relieves users from building complex client/server applications in order to access multiple data stores
- Innovation
  - Adds value by allowing arbitrary code/native query to be embedded
    - To preserve the expressivity of each data store's query mechanism
  - Provision for traditional query optimization with SQL and NoSQL data stores