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

dave@vcdave.com

blogs.forbes.com/davefeinleib

NOSQL (Not Only SQL)

- Specific DBMS, for web-based data
  - Specialized data models
    - Principle: No one size fits all
    - Key-value, table, document, graph
  - Trade relational DBMS properties
    - Full SQL, ACID transactions, data independence
  - For
    - Simplicity (schemaless, basic API)
    - Scalability and performance
    - Flexibility for the programmer (integration with programming language)

- NB: SQL is just a language and has nothing to do with the story

NoSQL Approaches

- Characterized by the data model, in increasing order of complexity:
  1. key-value: DynamoDB, Cassandra, Voldemort
  2. big table: Bigtable, Haddop Hbase, Accumulo
  3. document: 10gen MongoDB, Expresso
  4. graph: Neo4J, Pregel, Sparklee

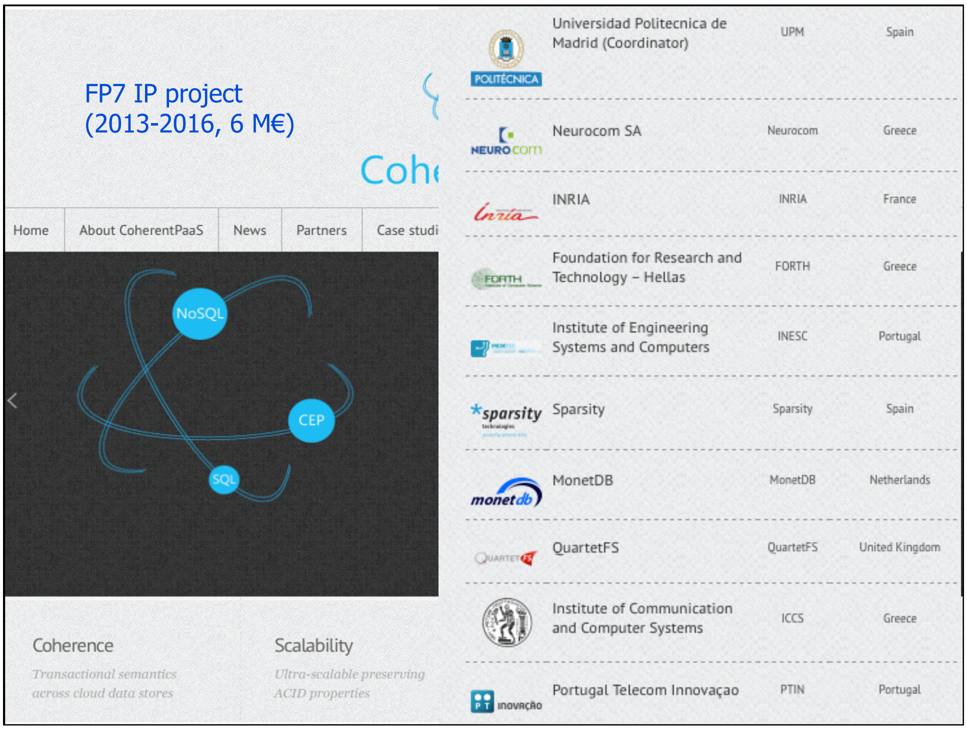
- What about object DBMS or XML DBMS?
  - Were there much before NoSQL
  - Sometimes presented as NoSQL
  - But not designed for scaling

NoSQL versus Relational

- The techniques are not new
  - Database machines, shared-nothing cluster
  - But very large scale
- Pros NoSQL
  - Scalability, performance
  - APIs suitable for programmers
- Pros Relational
  - Strong consistency, transactions
  - Standard SQL, many tools (OLAP cubes, BI, etc.)
- Towards NoSQL/Relational hybrids?
  - Google F1: "combines the scalability, fault tolerance, transparent sharding, and cost benefits so far available only in NoSQL systems with the usability, familiarity, and transactional guarantees expected from an RDBMS"

Outline

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



CloudMdsQL Objectives

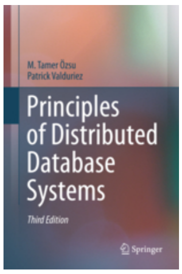
- Design an SQL-like query language to query multiple databases (SQL, NoSQL) in a cloud
  - Autonomous databases
    - This is different from recent multistore systems such as MISO (no autonomy)
- Design a query engine for that language
  - Compiler/optimizer
    - To produce an execution plan
  - Query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

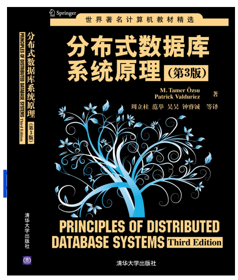
Issues

- No standard in NoSQL
  - Many different systems
    - Key-value store, big table, document, graph
- 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
  - Guided by the CoherentPaaS project uses cases

Related Work

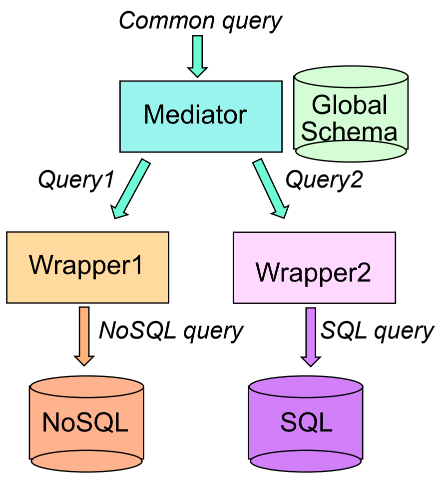
- Distributed multidatabase systems (or federated database systems)
  - A few databases (e.g. less than 10)
    - Corporate DBs
  - Powerful queries (with updates)
- Web data integration systems
  - Many data sources (e.g. 1000's)
    - DBs or files behind a web server
  - Simple queries (read-only)
- Dominant architecture is mediator/wra





Mediator/wrapper Architecture

- Mediator
  - Centralizes the information provided by the wrappers in a global schema
  - Transforms queries expressed in the common language into queries for the wrappers
  - Integrates the queries' results
- Wrapper
  - Exports information about the source schemas, and mapping functions that translate between source schemas and the mediator's schema
  - Transforms queries expressed in the common language into queries for the DBs
  - Transforms the queries' results in the common data model



Common Data Model and QL

- Major impact on data integration
  - Effectiveness, quality
- Main industry solutions
  - Relational/SQL
    - Simple data representation (tables) but rigid schema support
    - SQL familiar to users and developers, with SQL APIs used by many tools
  - XML/Xquery
    - Tree-based representation appropriate for Web data, which are typically semi-structured, and flexible schema capabilities
    - Xquery a complete, but complex language
  - JSON
    - Simpler than XML/Xquery but no standard QL
      - Many different languages (JSONpath, JSONiq, JAQL)

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 (but we can represent a graph with relations)
- 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*
  - 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*

- Key-value

**Any resemblance to living persons is coincidental*

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

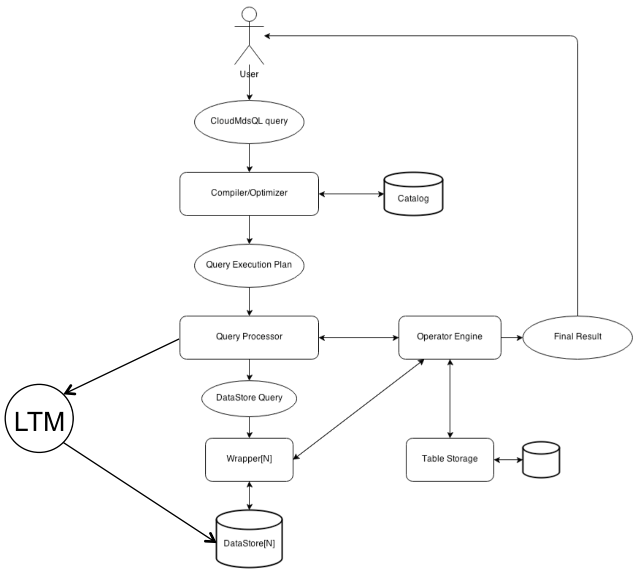
- Relational

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." }] })

Basic Query Engine Architecture



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

- It supports all data types from the common data model (including Null values)
- Many DBMSs have Python APIs (including Sparksee, MongoDB, MonetDB)
- It is simple, fairly well-known and easy to use
- It is rich in standard libraries
- Its interpreter is easily embeddable in other applications
- It is easy to wrap any (object-oriented or just procedural) API in Python without loss of functionality

Query Language

- Named table expression
  - Expression that returns a table representing a nested querie [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
  - Python named tables
    - Embedded blocks of Python statements that produce relations

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: Derby (Apache)
  - Document: MongoDB
  - Graph: Sparksee (Sparcity)

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.)

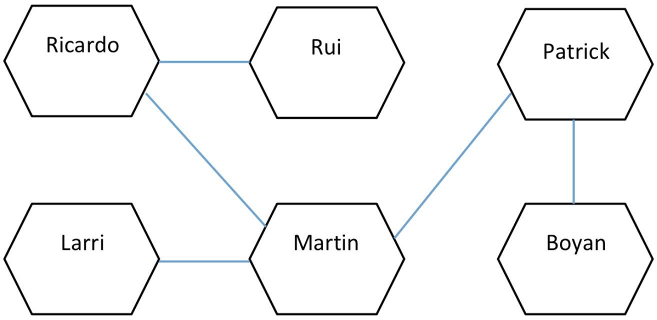
DB2: a document DB
Reviews (PID string, reviewer string, date string, review string)

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, …)



Q1: relational & document data

- Retrieve all publications from INRIA, reviewed in 2013.

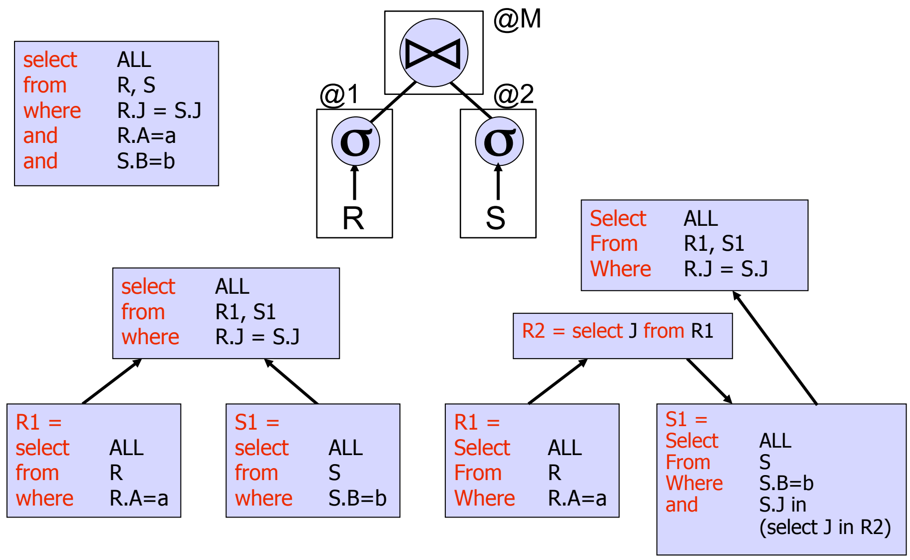
```
/* retrieve from document db the reviews made in 2013
reviews_2013( pub_id int, reviewer string )@DB2 = {*
  db.reviews.find(
    {'date': {'$gte': '2013-01-01', '$lte': '2013-12-31'} },
    {'pub_id': 1, 'reviewer': 1, '_id': 0} )
*}

/* retrieve from relational db the publications of scientists from Inria
pubs_I( id int, title string, author string )@DB1 = (
  SELECT pubs.id, pubs.title, pubs.author
  FROM pubs
    JOIN scientists ON pubs.author = scientists.name
  WHERE scientists.affiliation = 'INRIA'
)

/* join the two intermediate datasets
SELECT pubs_I.id, pubs_I.title, pubs_I.author, reviews_2013.reviewer
FROM pubs_I
  JOIN reviews_2013 ON pubs_I.id = reviews_2013.pub_id;
```

Id	Title	Author	Reviewer
5	Principles …	Patrick	Ricardo
5	Principles …	Patrick	Rui

Optimization with Bindjoin



Q2: Q1 with Bindjoin

```
/* Same as Q1
reviews_2013( pub_id int, reviewer string )@DB2 = {*
  db.reviews.find(
    {'date': {'$gte': '2013-01-01', '$lte': '2013-12-31'} },
    {'pub_id': 1, 'reviewer': 1, '_id': 0} )
*}

/* Retrieve only those records that match the join criteria
pubs_I( id int, title string, author string )@DB1 = (
  SELECT pubs.id, pubs.title, pubs.author
  FROM pubs
    JOIN scientists ON pubs.author = scientists.name
  WHERE scientists.affiliation = 'INRIA'
  AND pubs.id IN (SELECT pub_id FROM reviews_2013)
)

/* Same as Q1
SELECT pubs_I.id, pubs_I.title, pubs_I.author,
reviews_2013.reviewer
FROM pubs_I
  JOIN reviews_2013 ON pubs_I.id = reviews_2013.pub_id;
```

Query 3 on the 3 Databases

- Discover conflicts of interest in publications from Inria reviewed in 2013

```
reviews_2013( pub_id int, reviewer string )@DB2 = {*
  db.reviews.find(
    {'date': {'$gte': '2013-01-01', '$lte': '2013-12-31'} },
    {'pub_id': 1, 'reviewer': 1, '_id': 0} )
*}
pubs_I( id int, title string, author string )@DB1 = (
  SELECT pubs.id, pubs.title, pubs.author
  FROM pubs
    JOIN scientists ON pubs.author = scientists.name
  WHERE scientists.affiliation = 'INRI
  AND pubs.id IN (SELECT pub_id FROM r
)
conflicts( author string, reviewer string, conflict string
  JOINED ON author, reviewer )@DB3 =
{*
  for (A, R) in CloudMdsQL.Outer:
    sp = graph.FindShortestPathByName( A, R, max_hops=2 )
    if sp.exists():
      yield (A, R, 'Friend' + (sp.get_cost()-1) * 'OfFriend')
*}
SELECT p.id, p.title, p.author, r.reviewer, c.conflict
FROM pubs_I p
  JOIN reviews_2013 r ON p.id = r.pub_id
  JOIN conflicts c ON p.author = c.author AND r.reviewer = c.reviewer;
```

Id	Author	Reviewer	Conflict
5	Patrick	Ricardo	FriendOfFriend

Future work

- Query compiler
  - Add query language enhancements (parametrized expressions, stored expressions, etc.)
- Query engine
  - Efficient intermediate table management
- Query optimization
  - Query rewriting, e.g. Q1 => Q2
  - Cost model, e.g. to choose the best between Q1 and Q2
- Validation
  - With more NoSQL and SQL databases