Design and Implementation of the CloudMdsQL Multistore System

Patrick Valduriez
Inria, Montpellier, France

CoherentPaaS

With Boyan Kolev, Carlyna Bondiombouy, Oleksandra Levchenko, Ricardo Jimenez, Raquel Pau and Jose Pereira
Cloud & Big Data Landscape

Easy to get lost
No "one size fits all"
No standard
Keeps evolving

Copyright © 2012 Dave Feinleib
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
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
• Query language
• Query rewriting
• Use case example
• Experimental validation
CoherentPaaS

FP7 IP project (2013-2016, 6 M€)

CoherentPaaS

- full ACID coherent
- scalable
- NoSQL
- SQL
- CEP

Coherence
- Transactional semantics across cloud data stores

Scalability
- Ultra-scalable preserving ACID properties

Simplicity
- Programming with a single query language

Efficiency
- Avoiding ETLs (copying TBs of data across data stores)
FP7 IP project
(2013-2016, 6 M€)
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
Related Work (cont.)

- **Multistore systems**
  - Provide integrated access to multiple, heterogeneous cloud data stores such as NoSQL, HDFS and RDBMS
    - E.g. BigDAWG, BigIntegrator, Estocada, Forward, HadoopDB, Odyssey, Polybase, QoX, Spark SQL, etc.
  - Great for integrating structured (relational) data and big data
  - But typically trade data store autonomy for performance or work only for certain categories of data stores (e.g. RDBMS and HDFS)
Centralized Query Engine

User Application

CloudMdsQL Query

JDBC Client

Query Mediator

Execution Engine

Query Processor

Wrapper DS₁

DS₁

Wrapper DS₂

DS₂

Table Store Connector

Table Store
Centralized Query Engine

- Query engine
- Wrapper
- DS
- Table Store
- Connector
- Store

Straightforward M/W architecture

⇒

High communication cost DS – QE

Little optimization opportunities
Distributed Query Engine

User/App/GUI
CloudMdsQL client

Query

Master

Catalog

Cardinalities
Selectivities
Indexes

Query plan

Communication processor

Worker

Communication processor

Query plan

Data

Query Execution Controller

Operator Engine

Wrapper

Table Storage

DataStore

Worker

Communication processor

Query plan

Data

Query Execution Controller

Operator Engine

Wrapper

Table Storage

DataStore
Distributed Query Engine

Fully distributed architecture => Many optimization opportunities
CloudMdsQL Objectives

• Design an SQL-like query language to query multiple databases (SQL, NoSQL) in a cloud
  • While preserving the autonomy of the data stores
    • This is different from most 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: Derby, Sparksee, MongoDB, Hbase, MonetDB, Spark/HDFS, 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
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**\(^1,2\)
  - 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
  - Functions can transform types and do data-metadata conversion

---

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**

  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." } ] })

*Any resemblance to living persons is coincidental
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)

```sql
/* 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

\[ T_1(x \text{ int}, y \text{ int})@DB1 = ( \text{SELECT } x, y \text{ FROM } A ) \]

\[ T_2(x \text{ int}, z \text{ string})@DB2 = {*}\]
\[ \text{db.B.find( \{\$lt: \{x, 10\}\}, \{x:1, z:1, _id:0\} )} \]*

\[ \text{SELECT } T_1.x, T_2.z \]
\[ \text{FROM } T_1, T_2 \]
\[ \text{WHERE } T_1.x = T_2.x \text{ AND } T_1.y \leq 3 \]

\[
\begin{align*}
\pi_{x, z} & \quad \sigma_{T1.y \leq 3} \\
\pi_{x, y} & \quad X_1@DB1 (\text{Derby}) \\
\pi_x & \quad X_2@DB2 (\text{MongoDB}) \\
N_{x, z} & \quad \sigma_{y \leq 3} \\
X & \quad A \\
N_x, z & \quad X_2@DB2 (\text{MongoDB}) \\
\end{align*}
\]

\[ \text{SELECT } x \text{ FROM } A \text{ WHERE } y \leq 3 \]
Optimization with Bindjoin

```
select ALL from R, S
where R.J = S.J
and R.A=a
and S.B=b

R1 = select ALL from R
where R.A=a

S1 = select ALL from S
where S.B=b

R2 = select J from R1

R1 = Select ALL From R
Where R.A=a

S1 = Select ALL From S
Where S.B=b
and (select J in R2)
```
Sub-query Rewriting: bindjoin

T1(id int, x string)@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
Sub-query Rewriting: bindjoin

\[ \pi_{x, y} \]

\begin{array}{|c|c|}
\hline
x & y \\
\hline
abc & 1 \\
xyz & 9 \\
\hline
\end{array}

\[ \pi_{id, x} \]

<table>
<thead>
<tr>
<th>id</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>abc</td>
</tr>
<tr>
<td>3</td>
<td>xyz</td>
</tr>
</tbody>
</table>

\[ \pi_{id, y} \]

<table>
<thead>
<tr>
<th>id</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
</tbody>
</table>

\[ \text{SELECT id, } x \text{ FROM A} \]

\[ \text{SELECT id, } y \text{ FROM B WHERE id IN (1, 3)} \]
Use Case Bibliographic App. Example

- 3 data stores
  - Relational
  - Document
  - Graph
- A query that involves integrating data from the three data stores
Example DBs

DB1: a relational DB

Table Scientists (Name char(20), Affiliation char(10), Country char(30))

<table>
<thead>
<tr>
<th>Name</th>
<th>Affiliation</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ricardo</td>
<td>UPM</td>
<td>Spain</td>
</tr>
<tr>
<td>Martin</td>
<td>CWI</td>
<td>Netherlands</td>
</tr>
<tr>
<td>Patrick</td>
<td>INRIA</td>
<td>France</td>
</tr>
<tr>
<td>Boyan</td>
<td>INRIA</td>
<td>France</td>
</tr>
<tr>
<td>Larri</td>
<td>UPC</td>
<td>Spain</td>
</tr>
<tr>
<td>Rui</td>
<td>INESC</td>
<td>Portugal</td>
</tr>
</tbody>
</table>
Example DBs (cont.)

DB2: a document DB (MongoDB with SQL interface)

Document collection: publications

{id: 1, title: 'Snapshot Isolation', author: 'Ricardo', date: '2012-11-10'},
{id: 5, title: 'Principles of DDBS', author: 'Patrick', date: '2011-02-18'},
{id: 8, title: 'Fuzzy DBs', author: 'Boyan', date: '2012-06-29'},
{id: 9, title: 'Graph DBs', author: 'Larri', date: '2013-01-06'}

Document collection: reviews

{pub_id: “1”, reviewer: “Martin”, date: “2012.11.18”, review: “… text …”},
{pub_id: “5”, reviewer: “Rui”, date: “2013.02.28”, review: “… text …”},
{pub_id: “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, ...)

[Graph diagram showing relationships between Ricardo, Rui, Patrick, Larri, Martin, and Boyan]
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

- Retrieve scientists from INRIA
  @DB1 (MonetDB)

- Retrieve publications reviewed in 2013 and their reviewers
  @DB2 (MongoDB)

- Retrieve one- or two-level friendships by invoking BreadthFirstSearch()
  @DB3 (Sparksee)
CloudMdsSQL Query: expression

```python
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 CloudMdsSQL.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

\( \pi \) id, author, reviewer, friendship

\( \sigma \) year(review_date)=2013 AND affiliation='INRIA'

\( \pi \) name = author

\( + \text{bind} \) id, author, reviewer, friendship

\( \pi \) name

\( \pi \) id=pub_id

\( \pi \) id, title, author

\( \pi \) pub_id, reviewer

\( \eta \) person1, person2, friendship

\( \eta \) scientists

\( \eta \) publications

\( \eta \) reviews

\( @DB1 \) (MonetDB)

\( @DB2 \) (MongoDB)

\( @DB3 \) (Sparksee)
Rewritten Query Plan

```
SELECT name FROM scientists
WHERE affiliation = 'INRIA'
```

```
db.publications.find( {author: {$in: ['Patrick', 'Boyan']}} )
```

```
db.reviews.find({date: ...})
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Author</th>
<th>Reviewer</th>
<th>Friendship</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Patrick</td>
<td>Ricardo</td>
<td>friend-of-friend</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Author</th>
<th>Reviewer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patrick</td>
<td>Ricardo</td>
<td></td>
</tr>
<tr>
<td>Boyan</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Author</th>
<th>Reviewer</th>
<th>Friendship</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Patrick</td>
<td>Ricardo</td>
<td>friend-of-friend</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Author</th>
<th>Reviewer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patrick</td>
<td>Rui</td>
</tr>
<tr>
<td>Patrick</td>
<td>Ricardo</td>
</tr>
</tbody>
</table>

@DB1 (MonetDB)

@DB2 (MongoDB)

@DB3 (Sparksee)

@CloudMdsQL
Experimental Validation

• **Goal:** show the ability of the query engine to optimize CloudMdsQL queries

• **Prototype**
  • Compiler/optimizer implemented in C++ (using the Boost.Spirit framework)
  • Operator engine (C++) based on the query operators of the Derby 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

• **Deployment on a GRID5000 cluster**

• **Variations of the Bibliographic use case with 3 data stores**
  • Relational: Derby
  • Document: MongoDB
  • Graph: Sparksee
Experiments

- Variations of the Bibliographic use case with 3 data stores
  - Relational: Derby
  - Document: MongoDB
  - Graph: Sparksee

- Catalog
  - Information collected through the Derby and MongoDB wrappers
    - Cardinalities, selectivities, indexes

- 5 queries in increasing level of complexity
  - 3 QEPs per query
Experimental Results

**Query 1**

QEP\(_{11}\): \( \sigma_{QER}(R) \bowtie_3 P \)

QEP\(_{12}\): \( \sigma(R) \bowtie_3 P \)

QEP\(_{13}\): \( \sigma(R) \bowtie_3 P \)

**Query 2**

QEP\(_{21}\): \( (\sigma(S) \bowtie_1 P) \bowtie_1 \sigma(R) \)

QEP\(_{22}\): \( (\sigma(S) \bowtie_2 P) \bowtie_2 \sigma(R) \)

QEP\(_{23}\): \( (\sigma(S) \bowtie_2 P) \bowtie_3 \sigma(R) \)

**Query 3**

QEP\(_{31}\): \( ((\sigma(Sr) \bowtie_3 R) \bowtie_3 P) \bowtie_3 \sigma(Sa) \)

QEP\(_{32}\): \( ((\sigma(Sa) \bowtie_2 P) \bowtie_3 R) \bowtie_3 \sigma(Sr) \)

QEP\(_{33}\): \( (\sigma(Sa) \bowtie_2 P) \bowtie_3 (\sigma(Sr) \bowtie_3 R) \)
Experiment Results (cont.)

**Query 4**
QEP\(_{41}\) : (((\(\sigma\) (Sr) \(\bowtie\) \(_{3}\) R) \(\bowtie\) \(_{3}\) P) \(\bowtie\) \(_{3}\) F) \(\bowtie\) \(_{3}\) \(\sigma\) (Sa)
QEP\(_{42}\) : (((\(\sigma\) (Sa) \(\bowtie\) \(_{2}\) P) \(\bowtie\) \(_{3}\) R) \(\bowtie\) \(_{3}\) F) \(\bowtie\) \(_{3}\) \(\sigma\) (Sr)
QEP\(_{43}\) : ((\(\sigma\) (Sa) \(\bowtie\) \(_{2}\) P) \(\bowtie\) \(_{3}\) (\(\sigma\) (Sr) \(\bowtie\) \(_{3}\) R)) \(\bowtie\) \(_{3}\) F

**Query 5**
QEP\(_{51}\) : (((\(\sigma\) (Sr) \(\bowtie\) \(_{3}\) R) \(\bowtie\) \(_{3}\) P) \(\bowtie\) \(_{3}\) F) \(\bowtie\) \(_{3}\) \(\sigma\) (Sa)
QEP\(_{52}\) : (((\(\sigma\) (Sa) \(\bowtie\) \(_{2}\) P) \(\bowtie\) \(_{3}\) R) \(\bowtie\) \(_{3}\) F) \(\bowtie\) \(_{3}\) \(\sigma\) (Sr)
QEP\(_{53}\) : ((\(\sigma\) (Sa) \(\bowtie\) \(_{2}\) P) \(\bowtie\) \(_{3}\) (\(\sigma\) (Sr) \(\bowtie\) \(_{3}\) R)) \(\bowtie\) \(_{3}\) F
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 distributed query optimization with SQL and NoSQL data stores
References


