The CloudMdsQL Multistore System

Patrick Valduriez
Inria, Montpellier, France
Cloud & Big Data Landscape

Vertical Apps
- Predictive Policing
- bloomreach
- MYRRIX

Ad/Media Apps
- rocketfuel
- collective
- Recorded Future
- Bluefin
- DataXu

Data Processing Frameworks
- Spark
- Hadoop
- Hbase

NoSQL Databases
- Cassandra
- Couchbase
- 10gen

Analytics and Visualization
- Tableau
- Palantir

Data As A Service
- factual
- Gnip
- INRIX
- LexisNexis
- Locate

Operational Infrastructure
- Teradata
- MarkLogic
- Informatica

Infrastructure As A Service
- Amazon Web Services
- Windows Azure
- Google BigQuery

Structured Databases
- MySQL
- PostgresSQL

Copyright © 2012 Dave Feinleib
dave@vcdave.com
blogs.forbes.com/davefeinleib
Cloud & Big Data Landscape

Vertical Apps
- Predictive Policing
  - bloomreach
  - myrrh

Ad/Media Apps
- rocketfuel
- collective
- Recorded Future

Business Intelligence
- ORACLE
- Hyperion
- SAP
- BusinessObjects
- IBM

Analytics and Visualization
- tableau
- metaLaser
- MARKETS
- A*STER
- TIBCO
- karmasphere

NoSQL Databases
- MongoDB
-assandra
- Redis

Data Processing Frameworks
- Spark
- hadoop
- MapReduce

No standard
- Keeps evolving

Easy to get lost
- No "one size fits all"

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
- **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
- MFR statement
- 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€)

<table>
<thead>
<tr>
<th>Coherence</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Coherence</strong></td>
<td><strong>Scalability</strong></td>
</tr>
<tr>
<td>Transactional semantics across cloud data stores</td>
<td>Ultra-scalable preserving ACID properties</td>
</tr>
</tbody>
</table>

| 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 Innovacao | 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**
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)
First Try: centralized query engine

Query engine

- Query Processor
- Execution Engine
- Wrapper DS₁
- Wrapper DS₂
- DS₁
- DS₂
- Table Store

User Application → JDBC Client → Query Mediator → Execution Engine → Wrapper DS₁ → DS₁

CloudMdsQL Query

→ JDBC Client

→ Query Mediator

→ Execution Engine

→ Wrapper DS₂

→ Table Store Connector

→ Table Store
First Try: centralized query engine

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

**Straightforward M/W architecture**

- High communication cost DS – QE
- Little optimization opportunities
Second Try: distributed query engine
Second Try: 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
  • Guided by the CoherentPaaS project uses cases
    • E.g. bibliography search
Schema Issue: on read vs on write

- **Schema on write** (RDBMS, DW)
  - Prescriptive data modelling
    - Create schema S
    - Write data in S format
    - Query data in S format
  - Must change S before adding new data
  - Efficient querying but difficult evolution

- **Schema on read** (Hadoop, data lake)
  - Descriptive data modelling
    - Write data in native format
    - Create schema S
    - Query data in native format and transform to S (ETL on the fly)
  - One can add new data at anytime
  - Agility and flexibility, but less efficient querying
Our Design Choices

- **Data model: schema on read, table-based**
  - With rich data types
    - To allow computing on typed values
  - No global schema to define
    - Schema mapping within queries

- **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
  - 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." } ] })
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} )
*}
CloudMdsQL Optimization

• Query rewriting using
  • Select pushdown
  • Bindjoin
  • Join ordering
Select@ Pushdown Example

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

```
π
\[ x, z \]

σ
\[ T1.y \leq 3 \]

@CloudMdsQL

π
\[ x, y \]

T1@DB1
( MONETDB )

N
\[ x, z \]

T2@DB2
( MONGODB )

@CloudMdsQL

π
\[ x \]

σ
\[ y \leq 3 \]

A

T1@DB1
( MONETDB )

T2@DB2
( MONGODB )

SELECT x FROM A WHERE y <= 3
```
Bindjoin (recall)

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

select ALL from R1, S1
where R.J = S.J

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

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

R2 = select J from R1

S1 = /* semijoin */
Select ALL
From R1, S1
Where R.J = S.J

R1 = Select ALL
From R
Where R.A=a
and S.J in (select J in R2)
```
**Bindjoin Example**

```sql
SELECT T1.x, T2.y
FROM T1 BIND JOIN T2 ON T1.id = T2.id

T1(id int, x string)@DB1 = (SELECT id, x)
T2(id int, y int)@DB2 = (SELECT id, y FROM R2 )
```
Bindjoin Example

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>1</td>
</tr>
<tr>
<td>xyz</td>
<td>9</td>
</tr>
</tbody>
</table>

SELECT id, x FROM A

T1@DB1 (Derby)

<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>

SELECT id, y FROM B WHERE id IN (1, 3)

T2@DB2 (MonetDB)

<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>
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, …)

- Ricardo
- Rui
- Patrick
- Larri
- Martin
- 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 one- or two-level friendships by invoking BreadthFirstSearch()

- 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()
CloudMdsQL Query: expression

```sql
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

\[ \pi \text{id, author, reviewer, friendship} \]
\[ \sigma \text{year(review\_date) = 2013 AND affiliation='INRIA'} \]
\[ (\text{author, reviewer}) = (\text{person1, person2}) \]

\[ \pi \text{name = author} \]

\[ \pi \text{name} \]
\[ \text{scientists} @DB1 (MonetDB) \]

\[ \pi \text{id=pub_id} \]
\[ \text{publications} @DB2 (MongoDB) \]

\[ \pi \text{id, title, author} \]
\[ \pi \text{pub_id, reviewer} \]

\[ \text{reviews} @DB2 (MongoDB) \]

\[ \text{friendships@DB3 (Sparksee)} \]

\[ N \text{person1, person2, friendship} \]
Rewritten Query Plan

$\text{SELECT name FROM scientists WHERE affiliation = 'INRIA'}$

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

$\text{db.reviews.find({date: ... })}$
MFR Statement

• Sequence of Map/Filter/Reduce operations on datasets
  • Example: count the words that contain the string ‘cloud’

\[
\text{Dataset} \quad \text{SCAN(TEXT,'words.txt').MAP(KEY,1).FILTER( KEY LIKE '%cloud%' ).REDUCE (SUM)}
\]

• A dataset is an abstraction for a set of tuples, a Spark RDD
  • Consists of key-value tuples
  • Processed by MFR operations
MFR Example

- Query: retrieve data from RDBMS and HDFS

```sql
/* Integration subquery*/
SELECT title, kw, count FROM T1 JOIN T2 ON T1.kw = T2.word
WHERE T1.kw LIKE '%$cloud%'

/* SQL subquery */
T1(title string, kw string)@rdbms = ( SELECT title, kw FROM tbl )

/* MFR subquery */
T2(word string, count int)@hdfs = {
    SCAN(TEXT,'words.txt')
    .MAP(KEY,1)
    .REDUCE(SUM)
    .PROJECT(KEY,VALUE)
}
```
Query Rewriting

- Optimization techniques to reduce execution and communication costs
  - Selection pushdown
  - Performing bind join
  - MFR operators reordering and rewriting
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\(_{11}\): \( \sigma_{@QE}(R) \bowtie_3 P \)
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}: (((σ(Sr)⋈_3 R)⋈_3 P)⋈_3 F)⋈_3 σ(Sa)
QEP_{42}: (((σ(Sa)⋈_2 P)⋈_3 R)⋈_3 F)⋈_3 σ(Sr)
QEP_{43}: ((σ(Sa)⋈_2 P)⋈_3 (σ(Sr)⋈_3 R))⋈_3 F

Query 5
QEP_{51}: (((σ(Sr)⋈_3 R)⋈_3 P)⋈_3 F)⋈_3 σ(Sa)
QEP_{52}: (((σ(Sa)⋈_2 P)⋈_3 R)⋈_3 F)⋈_3 σ(Sr)
QEP_{53}: ((σ(Sa)⋈_2 P)⋈_3 (σ(Sr)⋈_3 R))⋈_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


BindJoin Optimization

- Challenge: how to apply bind join to any pair of data stores?
- 3 cases (for the right hand side, i.e., DS2)
  1. SQL support: easy!
  2. No SQL support but the datastore provides a powerful set-oriented query mechanism
  3. No SQL support and the data store provides only simple lookup
Case 2: set-oriented support

- DS2 has a set-oriented query mechanism (ActivePivot, Sparksee)
- The native query needs to access intermediate join keys from table storage
- Solution: add to the signature of S1 a clause to reference an intermediate table R1_keys
  - The join key values of R1 are provided in R1_keys and the native query for DS2 can use the mechanism that its wrapper provides to access these join keys

```c
/* Native subquery @ DS2 */
S1(B int, J int, COMMENT string JOINED ON J REFERENCING
   OUTER AS R1_keys )@DS2 =
(* native code for DS2 to perform the equivalent of the IN operator using R1_keys*)
```
Case 3: simple lookup

• DS2 provides only simple lookup (i.e. get (key) in a key-value store)
  • Solution: scalar lookup
    • Allows a parameterized named table (S1) to be used as a scalar function and evaluated for every value of a column from another table (R1)

    /* Native subquery @ DS2 */
    S1(B int, J int, COMMENT string WITHPARAMS J )@DS2 =
    {*
      get 'S_value', J
    *}

    • Then S1 is called in the SELECT list of the main SELECT statement of the query, instead of being joined with R1

    /* Integration query @ CloudMdsQL */
    SELECT R1.A, S1(J).B
    FROM R1