An Overview of Polystores

Patrick Valduriez

Inria, Montpellier, France

Joint work with *Boyan Kolev, Carlyna Bondiombouy, Oleksandra Levchenko and Ricardo Jimenez-Peris*

Cloud & Big Data Landscape

Easy to get lost

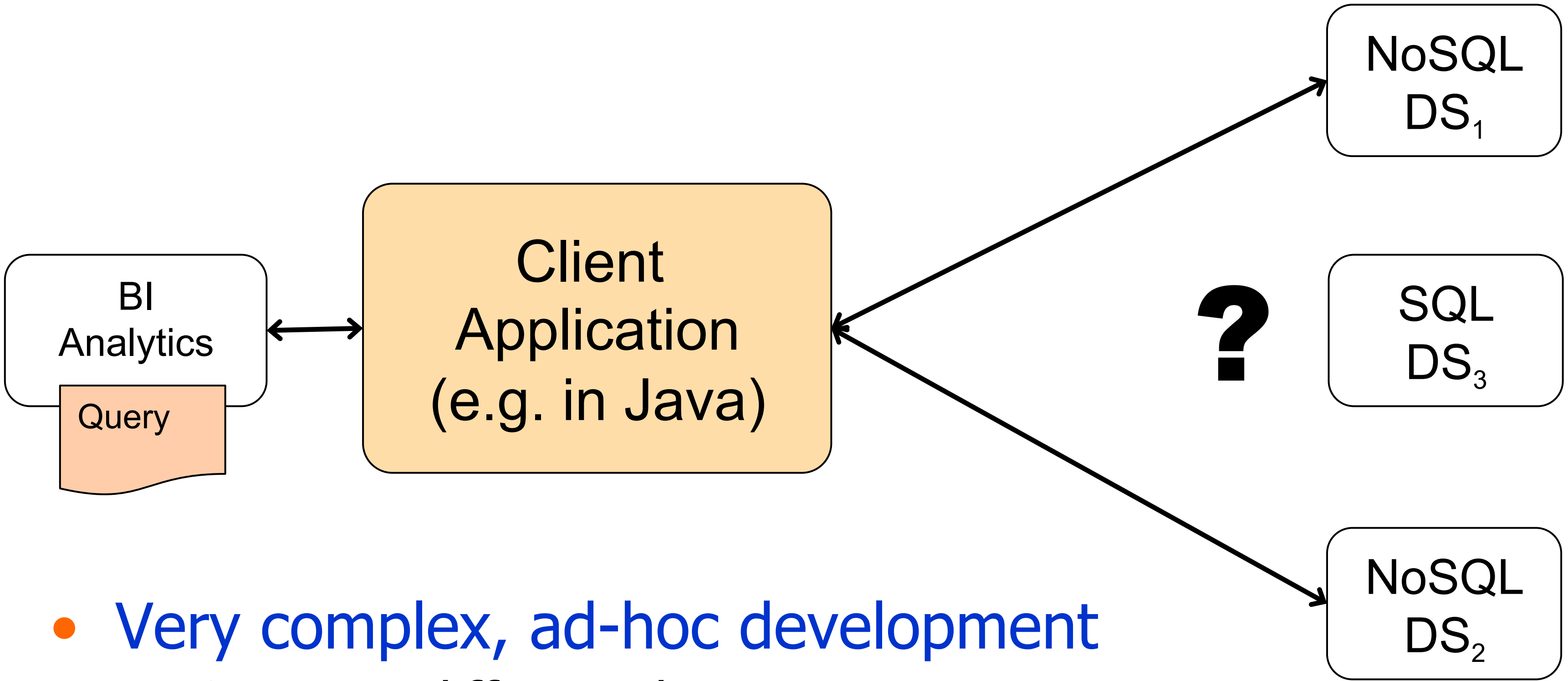
No "one size fits all"

No standard

Keeps evolving

 dave@vcdave.com blogs.forbes.com/davefeinleib

General Problem



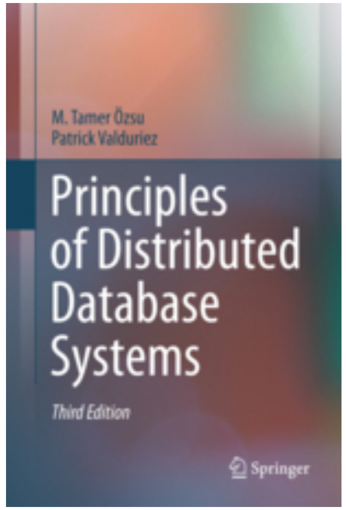
- Very complex, ad-hoc development
  - Querying different data sources
  - Managing intermediate results
  - Delivering (e.g. sorting) the final results
- Hard to extend
  - What if a new SQL DS appears?

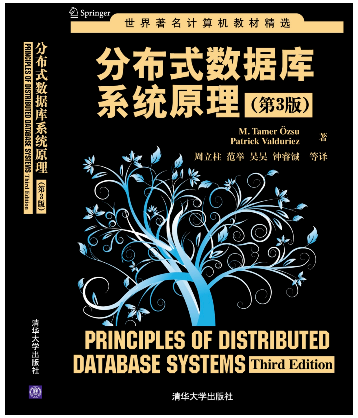
Outline

- Polystores
- The CloudMdsQL polystore
- Query language
- Distributed architecture
- Extending CloudMdsQL with MFR
- CloudMdsQL contributions

Origins of Polystores

- 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





Fourth Edition, 2018

Polystores

- Also called multistore systems
  - A major topic of research [The Case for Polystores. M. Stonebraker's blog. July 2015]
  - Provide integrated access to multiple, heterogeneous cloud data stores such as NoSQL, HDFS, CEP and RDBMS
  - 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)

Taxonomy of Polystores*

- Three kinds
  - Loosely-coupled
    - Similar to mediator/wrapper
    - Common interface
    - Autonomy of data stores, i.e. the ability to be locally controlled (independent of the multistore)
  - Tightly-coupled
    - Exploit local interfaces for efficiency
    - Trade data store autonomy for performance
      - Materialized views, indexes
  - Hybrid
    - Compromise between loosely- and tightly-coupled

*C. Bondiombouy, P. Valduriez. Query Processing in Cloud Multistore Systems: an overview. *Int. Journal of Cloud Computing*, 5(4): 309-346, 2016.

Comparisons: functionality

Polystore	Objective	Data model	Query language	Data stores
Loosely-coupled				
BigIntegrator (Uppsala U.)	Querying relational and cloud data	Relational	SQL-like	BigTable, RDBMS
Forward (UC San Diego)	Unyfing relational and NoSQL	JSON-based	SQL++	RDBMS, NoSQL
QoX (HP labs)	Analytic data flows	Graph	XML based	RDBMS, ETL
Tightly-coupled				
Polybase (Microsoft)	Querying Hadoop from RDBMS	Relational	SQL	HDFS, RDBMS
HadoopDB (Yale U.)	Querying RDBMS from Hadoop	Relational	SQL-live (HiveQL)	HDFS, RDBMS
Estocada (Inria)	Self-tuning	No common model	Native query languages	RDBMS, NoSQL
Hybrid				
SparkSQL (UCB)	SQL atop Spark	Nested	SQL-like	HDFS, RDBMS
BigDAWG (MIT)	Unifying relational and NoSQL	No common model	Island query languages, with CAST and SCOPE operators	RDBMS, NoSQL, Array DBMS, DSMSs

Comparisons: implementation

Polystore	Special modules	Schema mgt	Query processing	Query optimization
Loosely-coupled				
BigIntegrator (Uppsala U.)	Importer, absorber, finalizer	LAV	Access filters	Heuristics
Forward (UC San Diego)	Query processor	GAV	Data store capabilities	Cost-based
QoX (HP Labs)	Dataflow engine	No	Data/ function shipping, operation decomposition	Cost-based
Tightly-coupled				
Polybase (Microsoft)	HDFS bridge	GAV	Query splitting	Cost-based
HadoopDB (Yale U.)	SMS planer, dbconnector	GAV	Query splitting	Heuristics
Estocada (Inria)	Storage advisor	Materialized views	View-based query rewriting	Cost-based
Hybrid				
SparkSQL (UCB)	Catalyst extensible optimizer	Dataframes	In-memory caching using columnar storage	Cost-based
BigDAWG (MIT)	Island query processors	GAV within islands	Function/ data shipping	Heuristics

The CloudMdsQL Polystore*

- A hybrid polystore
  - Context: CoherentPaaS FP7 (2013-2016)

CoherentPaaS

- Objectives
  - Design an SQL-like query language to query multiple data sources in a cloud
    - Autonomous data stores
  - Design a query engine for that language
    - Fully distributed over a cluster's nodes
    - Compiler/optimizer
      - To produce efficient query execution plans
  - Design an ultra-scalable transaction manager

*B. Kolev, C. Bondiombouy, P. Valduriez, R. Jiménez-Peris, R. Pau, J. Pereira. The CloudMdsQL Multistore System. *SIGMOD 2016.*

The CloudMdsQL Language*

- Functional SQL-like query language
  - Can represent all query building blocks as functions
    - A function can be expressed in one of the DS languages, as a native function
      - E.g. a breadth-first search on a graph DS
  - Function results can be used as input to subsequent functions
  - Functions can transform types and do data-metadata conversion

*B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau, J. Pereira. CloudMdsQL: Querying Heterogeneous Cloud Data Stores with a Common Language. *Distributed and Parallel Databases*, 34(4): 463-503, 2016.

CloudMdsQL 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 with a schema on read
    - No need for global 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

CloudMdsQL Query Example

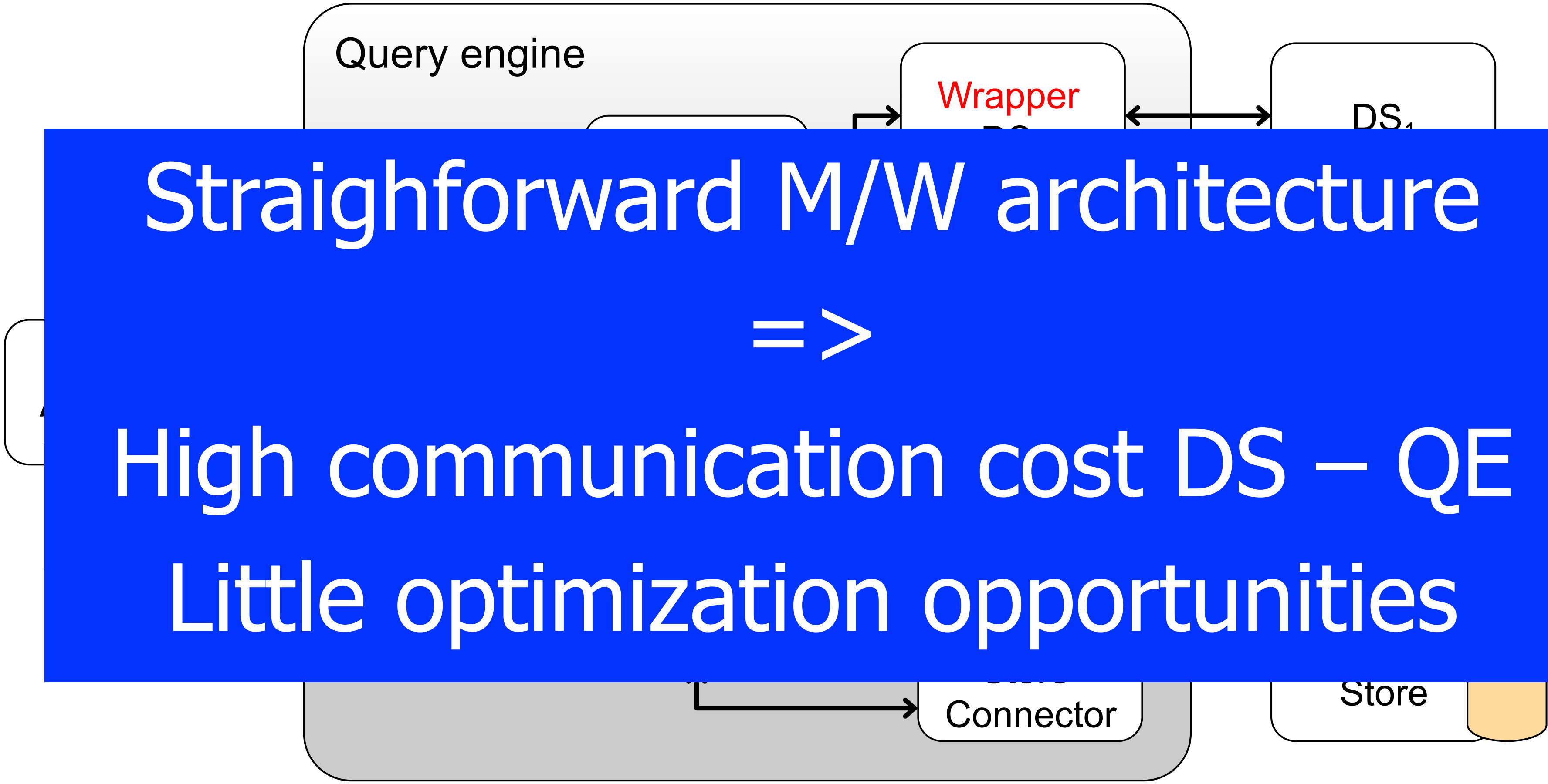
```
T1(x int, y int)@DS1 = ( SELECT x, y FROM A )

T2(x int, z string)@DS2 = {*
  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
```

SQL subquery on PostgreSQL

Native subquery on MongoDB

Integration subquery

Centralized Query Engine



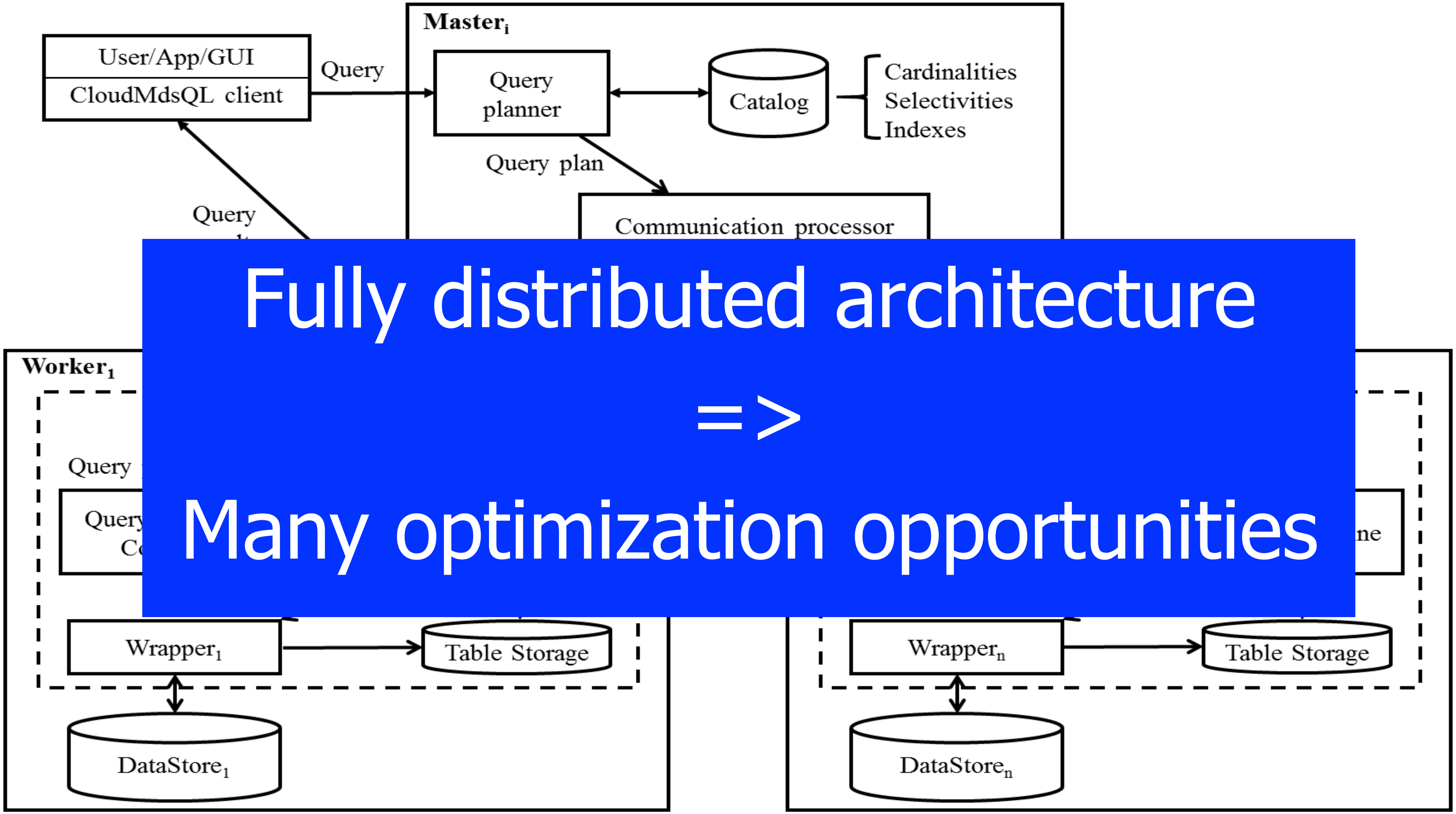
Straightforward M/W architecture

=>

High communication cost DS – QE

Little optimization opportunities

Distributed Query Engine



Fully distributed architecture

=>

Many optimization opportunities

Extending CloudMdsQL with MFR*

- Objectives
  - Integration of relational and HDFS data
    - With autonomy of data stores, unlike e.g. Polybase
  - Query data stored in HDFS using a data processing framework (DPF) like Spark or Flink
    - Using powerful functions lile Map, Filter, Reduce, etc.
- Issues
  - Execute joins between RDBMS and HDFS
  - Extend the CloudMdsQL Query Engine to work with Spark
- Solution
  - Map-Filter-Reduce (MFR) expression

*C. Bondiombouy, B. Kolev, O. Levchenko, P. Valduriez. Integrating Big Data and Relational Data with a Functional SQL-like Query Language. *DEXA 2015.* Extended version in Springer *TLDKS* journal 9940:48-74, 2016.

Motivating Example

An editorial office needs to find appropriate reporters for a list of publications based on given keywords

$\bowtie$

Publications (relational data)

title	author	kw
On cloud storage …	charlie	cloud
On cloud storage …	charlie	storage

After MapReduce processing

keyword	expert	freq
cloud	alice	2
storage	alice	1
virtual	bob	1
app	bob	1

Posts (HDFS)

2014-12-13, alice, storage, cloud
2014-12-22, bob, cloud, virtual, app
2014-12-24, alice, cloud

MFR Expression

- Works on a dataset, i.e. an abstraction for a set of tuples in a DPF
  - For instance, a Resilient Distributed Dataset in Spark
  - Consists of key-value tuples
- Using Map-Filter-Reduce operations
  - Map, Filter, Reduce : the main operations
  - Other operations : Scan, FlatMap, Project, …

MFR Expression Example

- Example: count the words that contain the string 'cloud'

Dataset

SCAN(TEXT,'words.txt') .MAP(KEY,1).FILTER(KEY LIKE '%cloud%').REDUCE (SUM)

Example Query with MFR

- Query: retrieve data from RDBMS and HDFS

```
/* 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)  *}

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

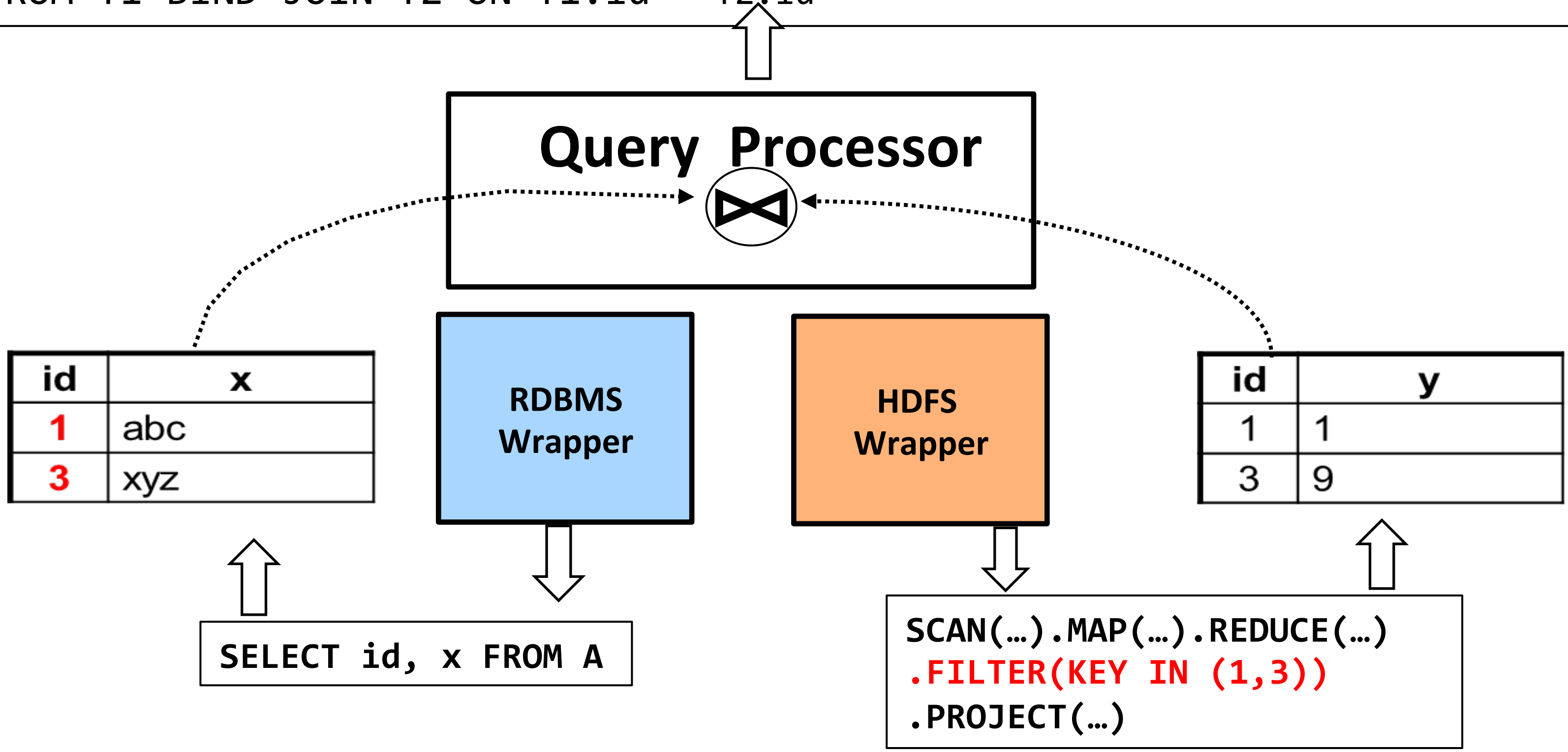
Query Optimization

- We apply known optimization techniques to reduce execution time and communication costs
  - Selection pushdown inside subqueries
  - Bind join
  - MFR operators reordering

Bind Join - example

```
T1(id int, x string)@DS1 = ( SELECT id, x FROM A ) /* SQL subquery */

T2(id int, y int)@DS2 = {                        /* DPF subquery */
        SCAN(…).MAP(…).RED                  EY, VALUE) *}

SELECT T1.x, T2.y                                /* integration subquery */
FROM T1 BIND JOIN T2 ON T1.id = T2.id
```

x	y
abc	1
xyz	9



MFR Rewrite Rules

- Rules for reordering MFR operators, based on their algebraic properties
- Focus on permuting FILTER with
  - PROJECT
  - REDUCE
  - MAP

Rule PROJECT / FILTER

PROJECT (<expr_list>).SELECT(<predicate1>)

=>

FILTER(<predicate2>).PROJECT(<expr_list>)

```
T1(a int, b int)@db1 = {* … .PROJECT (KEY, VALUE[0]) *}
SELECT a, b FROM T1 WHERE a > b
```

```
T1(a int, b int)@db1 ={* … .
FILTER(KEY>VALUE[0]).PROJECT(KEY,VALUE[0])*}
SELECT a, b FROM T1
```

Rule REDUCE / FILTER

REDUCE(<transformation>).FILTER(<predicate>)

=>

FILTER(<predicate>).REDUCE(<transformation>)

REDUCE (SUM) .FILTER(KEY LIKE ‘%cloud%)

FILTER (KEY LIKE ‘%cloud%’) . REDUCE (SUM)

Rule MAP / FILTER

MAP(<expr_list>).FILTER(<predicate1>)

=>

FILTER(<predicate2>).MAP(<expr_list>)

MAP(VALUE [0], KEY) .FILTER(KEY > VALUE)

FILTER (VALUE [0] > KEY) . MAP (VALUE [0], KEY)

Map/Filter/Reduce => Spark

- We need to translate MFR operators to Spark operators
  - map
  - flatMap
  - reduceByKey
  - aggregateByKey
  - filter

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
- Validation
  - With 10 different data stores, including SQL, NoSQL and Spark
  - Transfer to the Leanxcale startup