Multi-model Databases and Tightly Integrated Polystores
Current Practices, Comparisons, and Open Challenges

Jiaheng Lu, Irena Holubová, Bogdan Cautis
Outline

- Motivation and multiple model examples (30')
- Theoretical foundations (30')
- Multi-model data storage (25')
- Questions and discussion (5')

Session break

- Multi-model data query languages (10')
- Multi-model query processing (10')
- Overview on tightly integrated polystores (20')
- Query processing in tightly integrated polystores (15')
- Advanced aspects of tightly integrated polystores (15')
- Comparison of multi-model databases and tightly integrated polystores (5')
- Open problems and challenges (10')
- Questions and discussion (5')
A grand challenge on Variety

- **Big data**: Volume, Variety, Velocity, Veracity

- **Variety**: hierarchical data (XML, JSON), graph data (RDF, property graphs, networks), tabular data (CSV), etc
Motivation: E-commerce
Motivation: one application to include multi-model data

- Relational data: customer databases
- Graph data: social networks
- Hierarchical data: catalog, product
- Text data: Customer Review
- ......

An E-commerce example with multi-model data
Two solutions

1. Polystores
   Using jointly *multiple* data storage technologies, chosen based upon the way data is being used by individual applications.

1. Multi-model databases
   Using *one single, integrated* backend.
Polystores

• Use the right tool for (each part of) the job
• If you have structured data with some differences
  • Use a document store
• If you have relations between entities and want to efficiently query them
  • Use a graph database
• If you manage the data structure yourself and do not need complex queries
  • Use a key-value store

Glue everything together...
Multiple NoSQL databases
Pros and Cons of Polystores

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
</table>
| • Handle multi-model data  
• Help your apps to scale well  
• A rich experience | • Requires the company to hire people to integrate different databases  
• Implementers need to learn different databases  
• It is a challenge to handle cross-model query and transaction |
Three types of polystore systems*

- **Loosely-coupled systems**
  - Similar to mediator / wrapper
  - Common interfaces
  - Autonomy of local stores

- **Tightly-coupled systems**
  - Trade autonomy for performance with materialized views and indexes

- **Hybrid**
  - Compromise between loosely-coupled and tightly

---

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

MyriaL and Python

RACO Middleware
Translation, Optimization, Orchestration

MyriaX
Relational

Radish
C++ HPC cluster

SciDB
Array

Spark
Key-Value

SPARQL
RDF

Data Transfer with PipeGen

http://myria.cs.washington.edu/
Two solutions

1. Polystores
   Using jointly multiple data storage technologies, chosen based upon the way data is being used by individual applications.

1. Multi-model databases
   Using one single, integrated backend
Multi-model DB

- One unified database for multi-model data
Multi-model databases

- A multi-model database is designed to support multiple data models against a single, integrated backend.

- Document, graph, relational, and key-value models are examples of data models that may be supported by a multi-model database.
Multi-model databases: One size fits multi-data-model
Most of DBs became multi-model databases in 2017

• By 2017, all leading operational DBMSs will offer multiple data models, relational and NoSQL, in a single DBMS platform.

--- Gartner report for operational databases 2016
Three examples of multi-model databases

ORACLE®

OrientDB®

ArangoDB
Oracle database provides a long list of supported data models that can be used and managed inside Oracle database:

- JSON document
- Spatial and Graph Data
- XML DB data
- Text data
- Multimedia data
Example: Data transformation by views between JSON and relation data

<table>
<thead>
<tr>
<th>Tasks</th>
<th>SQL/JSON language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct JSON View from relational content</td>
<td>CREATE JSON_VIEW AS SELECT JSON {“Staff”: {“STAFF_ID”: e.staff_id, “First”: e.first, “Last”: e.last, “Mgr”: e.mgr, “Dept”: {“Dept_ID”: d.dept_id, “Names”: d.name, “Head”: d.head }}} FROM Employee e, department d WHERE e.dep_id = d.dep_id</td>
</tr>
<tr>
<td>Construct relational view of employee from JSON</td>
<td>CREATE EMPLOYEE_REL_VIEW AS SELECT * FROM JSON_VIEW f, JSON_TABLE (f.Staff) (Staff_ID, First, Last, Mgr)</td>
</tr>
</tbody>
</table>
ArangoDB is designed as a native multi-model database, supporting **key/value**, **document** and **graph** models.
An example of multi-model data and query

<table>
<thead>
<tr>
<th>Customer_ID</th>
<th>Name</th>
<th>Credit_limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mary</td>
<td>5,000</td>
</tr>
<tr>
<td>2</td>
<td>John</td>
<td>3,000</td>
</tr>
<tr>
<td>3</td>
<td>William</td>
<td>2,000</td>
</tr>
</tbody>
</table>

Mary (1)  
Friend  
William (3)  
John (2)  

"1" --> "34e5e759"  
"2" --> "0c6df508"

{"Order_no":"0c6df508",  
"Orderlines": [  
{ "Product_no":"2724f",  
"Product_Name":"Toy",  
"Price":66 },  
{ "Product_no":"3424g",  
"Product_Name":"Book",  
"Price":40 } ]}
An example of multi-model data and query

<table>
<thead>
<tr>
<th>Customer_ID</th>
<th>Name</th>
<th>Credit_limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mary</td>
<td>5,000</td>
</tr>
<tr>
<td>2</td>
<td>John</td>
<td>3,000</td>
</tr>
<tr>
<td>3</td>
<td>William</td>
<td>2,000</td>
</tr>
</tbody>
</table>

{"Order_no":"0c6df508",
 "Orderlines": [
  { "Product_no":"2724f",
  "Product_Name":"Toy",
  "Price":66 },
  { "Product_no":"3424g",
  "Product_Name":"Book",
  "Price":40 } ]}
Q: Return all products which are ordered by a friend of a customer whose credit limit is over 3000

<table>
<thead>
<tr>
<th>Customer_ID</th>
<th>Name</th>
<th>Credit_limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mary</td>
<td>5,000</td>
</tr>
<tr>
<td>2</td>
<td>John</td>
<td>3,000</td>
</tr>
<tr>
<td>3</td>
<td>William</td>
<td>2,000</td>
</tr>
</tbody>
</table>

Order lines:
- Order_no: 0c6df508
  - Product_no: 2724f, Product_Name: Toy, Price: 66
  - Product_no: 3424g, Product_Name: Book, Price: 40
Recommendation query:
Return all products which are ordered by a friend of a customer whose credit limit is over 3000.
• Supporting graph, document, key/value and object models.
• It supports schema-less, schema-full and schema-mixed modes. Queries with SQL extended for graph traversal.
Select expand(out("Knows").Orders.orderlines.Product_no) from Customers where CreditLimit > 3000

Recommendation query:
Return all products which are ordered by any friend of a customer whose credit limit is over 3000.
What is the difference between Multi-model and Multi-modal

• **Multi-model**: graph, tree, relation, key-value,…

• **Multi-modal**: video, image, audio, eye gaze data, physiological signals,…
Three arguments on multi-model data management

• 1. One size cannot fit all

• 2. One size can fit all

• 3. One size fits a bunch
One size cannot fit all

“SQL analytics, real-time decision support, and data warehouses cannot be supported in one engine.”

M. Stonebraker and U. Cetintemel. ”One Size Fits All”: An Idea Whose Time Has Come and Gone (Abstract). In ICDE, 2005.
One size can fit all

- OctopusDB suggests a unified, one size fits all data processing architecture for **OLTP, OLAP, streaming systems, and scan-oriented database systems.**

Jens Dittrich, Alekh Jindal: Towards a One Size Fits All Database Architecture. CIDR 2011: 195-198
One size can fit a bunch: AsterixDB

AsterixDB System Overview

Providing Hadoop-based query platforms, key-value stores and semi-structured data management

A simple survey

How many of you agree that (You can choose both or all or none of them)

1. One size cannot fit all

2. One size can fit all

3. One size fits a bunch

4. ???
Outline

- Motivation and multiple model examples (30')
- Theoretical foundations (30')
- Multi-model data storage (25')
- Questions and discussion (5')

Session break

- Multi-model data query languages (10')
- Multi-model query processing (10')
- Overview on tightly integrated polystores (20')
- Query processing in tightly integrated polystores (15')
- Advanced aspects of tightly integrated polystores (15')
- Comparison of multi-model databases and tightly integrated polystores (5')
- Open problems and challenges (10')
- Questions and discussion (5')
Theoretical foundation for multi-model management

Diagram to illustrate 2-category
Challenge: a new theory foundation

Call for a **unified model and theory** for multi-model data!

The theory of relations (150 years old) is not adequate to mathematically describe modern (NoSQL and multi-model) DBMS.
Two possible theoretical models

- Category theory

- Associative array
One possible theory foundation: category theory

- Introduced to mathematics world by Samuel Eilenberg and Sauders MacLane in 1944

- Found as part of their work in topology

- Category theory becomes the theoretical foundation on functional programming: Haskell
Categories Defined

● A category $C$ is ....
  ● a collection of objects $\text{ob}(C)$ .. \{X, Y, Z ....\}  
  ● a collection of morphisms \{f, g ....\}  
    ● A set of morphisms from object $a$ into $b$ is denoted by $\text{Hom}_c(a, b)$ or $a\to b$. 

\[ \begin{array}{ccc}
X & \xrightarrow{f} & Y \\
\downarrow{g \circ f} & & \downarrow{g} \\
& Z &
\end{array} \]
Categories Defined (con’t)

- The category must satisfy the following rules
  - associativity
    - \((h \circ g) \circ f = h \circ (g \circ f)\) \([a, b, c, d \in \text{ob}(C), f \in \text{Hom}_c(a, b), g \in \text{Hom}_c(b, c), h \in \text{Hom}_c(c, d)\]\n  - unit laws
    - \(f \circ 1_a = f = 1_b \circ f\)

- Think of it like a graph: the nodes are objects and the arrows are relationships
Relational category

- A relational category $C$
  - an $\text{ob}(C)$ is a table
  - a morphisms $a \rightarrow b$ means that $a$ has the relational homomorphism with $b$

Table A

<table>
<thead>
<tr>
<th>Staff_ID</th>
<th>First</th>
<th>Last</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>John</td>
<td>Smith</td>
</tr>
</tbody>
</table>

Table B

<table>
<thead>
<tr>
<th>Staff_ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>John Smith</td>
</tr>
<tr>
<td>101</td>
<td>James William</td>
</tr>
</tbody>
</table>
JSON category

- A JSON category $J$
  - an $ob(J)$ is a JSON file
  - a morphisms $a \to b$ means that $a$ has a **tree homomorphism** with $b$

**JSON A**

```json
{Staffs: 
  {"Staff_ID":"100","First":"John", "Name": "John Smith"},
  }
}
```

**JSON B**

```json
{Staffs: {
  "Staff_ID":"101", "First":"James", "Last":"William"}
}
```
Graph category

- A Graph category $G$
  - an $\text{ob}(J)$ is a graph
  - a morphisms $a \rightarrow b$ means that $a$ has a graph homomorphism with $b$

Graph A

- John
  - Dept
    - Sales
  - Manager
    - James

Graph B

- John
  - Dept
    - Sales
A single object can contain multi-model data

One object in a category contains both graph and table data.

<table>
<thead>
<tr>
<th>Customer_ID</th>
<th>Name</th>
<th>Credit_limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mary</td>
<td>5,000</td>
</tr>
<tr>
<td>2</td>
<td>John</td>
<td>3,000</td>
</tr>
<tr>
<td>3</td>
<td>William</td>
<td>2,000</td>
</tr>
</tbody>
</table>

Marry (1) Friend

William (3) Friend

John (2)
Product and Pull-back in categories

Product

Pull-back
An example of **Product**

<table>
<thead>
<tr>
<th>Name</th>
<th>Credit_limits</th>
<th>G.Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>5,000</td>
<td>Mary</td>
</tr>
<tr>
<td>John</td>
<td>3,000</td>
<td>William</td>
</tr>
<tr>
<td>William</td>
<td>2,000</td>
<td>William</td>
</tr>
</tbody>
</table>
An example of **Push-back**

<table>
<thead>
<tr>
<th>Name</th>
<th>Credit_limits</th>
<th>G.Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>5,000</td>
<td>Mary</td>
</tr>
<tr>
<td>John</td>
<td>3,000</td>
<td>John</td>
</tr>
<tr>
<td>William</td>
<td>2,000</td>
<td>William</td>
</tr>
</tbody>
</table>

**G**
- Friend
  - William
- Mary
- Friend
  - John
An example of multi-model data and query

**Table T**

<table>
<thead>
<tr>
<th>Customer_ID</th>
<th>Name</th>
<th>Credit_limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mary</td>
<td>5,000</td>
</tr>
<tr>
<td>2</td>
<td>John</td>
<td>3,000</td>
</tr>
<tr>
<td>3</td>
<td>William</td>
<td>2,000</td>
</tr>
</tbody>
</table>

**Graph G**

- Friend of Mary (1)
- Friend of John (2)
- Friend of William (3)

**JSON K**

```json
{"Order_no":"0c6df508",
 "Orderlines": [
  {
    "Product_no":"2724f",
    "Product_Name":"Toy",
    "Price":66
  },
  {
    "Product_no":"3424g",
    "Product_Name":"Book",
    "Price":40
  }
]}
```

"1" --> "34e5e759"
"2" --> "0c6df508"
Join with four models of data by Pull-back

Q: Return all products which are ordered by any friend of a customer whose credit_limit > 3000
Functors

- a “category of categories”
  - objects are categories, morphisms are mappings between categories
  - preserves identity and composition properties
Functors for data transformation

Table A

<table>
<thead>
<tr>
<th>Dep_ID</th>
<th>Name</th>
<th>Head</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2</td>
<td>Sales</td>
<td>101</td>
</tr>
</tbody>
</table>

Table B

<table>
<thead>
<tr>
<th>Dep_ID</th>
<th>Name</th>
<th>Head</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2</td>
<td>Sales</td>
<td>101</td>
</tr>
<tr>
<td>D3</td>
<td>Production</td>
<td>102</td>
</tr>
</tbody>
</table>

JSON A

```json
{Depts: {
"Dep_ID":"D2","Name":"Sales",
}}
```

JSON B

```json
{Depts: {
"Dep_ID":"D2","Name":"Sales",
"Head": "101"
}
{"Dep_ID":"D3","Name":"Production",
"Head": "102"}}
```
A natural transformation provides a way of transforming one functor into another.
Natural transformation example

**JSON A**

```javascript
{Depts: {
  "Dep_ID":"D2","Name":"Sales",
  Head: {
    "ID":"101","Name":"John"}
},
{Depts: "Dep_ID":"D3","Name":"Production",Head: {
  "ID":"102","Name":"James"}}
```

**Table A1**

<table>
<thead>
<tr>
<th>Dep_ID</th>
<th>Name</th>
<th>Head_ID</th>
<th>Head_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2</td>
<td>Sales</td>
<td>101</td>
<td>John</td>
</tr>
<tr>
<td>D3</td>
<td>Production</td>
<td>102</td>
<td>James</td>
</tr>
</tbody>
</table>

**Tables A2**

<table>
<thead>
<tr>
<th>ID</th>
<th>Head_ID</th>
<th>Head_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>H01</td>
<td>John</td>
</tr>
<tr>
<td>102</td>
<td>H02</td>
<td>Jame</td>
</tr>
</tbody>
</table>
Category theory: mathematical foundation for MMDB

1. **Category object**: an abstract definition of object in multi-model databases: including relation, tree, graph, key-value pair

2. **Query semantics**: product, pull-back, limits in category theory

3. **Proof of the equivalence of declarative and procedural syntaxes** over the above definitions: functor and natural transformation

4. **Proof of data instance equivalence** for multi-model data
Two possible theoretical model

- Category theory
- Associative array
Associative arrays

Associative arrays could provide a mathematical model for polystores to optimize the exchange of data and execution queries.

Definition: \( A: \{1, \ldots, m\} \times \{1, \ldots, n\} \rightarrow V \)
Associative array for relations and graphs

A(row1,Name)="Mary"
A(row2,Name)="John"
A(row3,Name)="William"

<table>
<thead>
<tr>
<th>Customer_ID</th>
<th>Name</th>
<th>Credit_limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mary</td>
<td>5,000</td>
</tr>
<tr>
<td>2</td>
<td>John</td>
<td>3,000</td>
</tr>
<tr>
<td>3</td>
<td>William</td>
<td>2,000</td>
</tr>
</tbody>
</table>

B(Edge1, start)="William" , B(Edge1, end)="Mary"
B(Edge2, start )="Mary"
B(Edge2, end)="John"
## Associative array example

<table>
<thead>
<tr>
<th>Customer_ID</th>
<th>Name</th>
<th>Credit_limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mary</td>
<td>5,000</td>
</tr>
<tr>
<td>2</td>
<td>John</td>
<td>3,000</td>
</tr>
<tr>
<td>3</td>
<td>William</td>
<td>2,000</td>
</tr>
</tbody>
</table>

Matrix A

\[
C = PB \oplus PP^T A
\]

\[
P = I_A (A \oplus \odot B^T) I_B
\]

\[
= I_A (A \land B^T) I_B
\]

Matrix B

- $\text{C(row3, start)="William"}$
- $\text{C(row1, end)="Mary"}$
- $\text{C(row2, end)="John"}$
From SQL to Associative algebra

\[
\text{SELECT } J(1), \ldots, J(n) \text{ FROM } A 
\quad \mapsto \quad A \cdot J(1), \ldots, J(n)
\]

\[
\text{SELECT } * \text{ FROM } A \text{ UNION SELECT } * \text{ FROM } B 
\quad \mapsto \quad A \oplus B
\]

\[
\text{SELECT } * \text{ FROM } A \text{ INTERSECT SELECT } * \text{ FROM } B 
\quad \mapsto \quad PB \quad \text{or} \quad P^T A
\]

\[
\text{SELECT } * \text{ FROM } A, B \text{ WHERE } \theta(A, J(1), \ldots, J(n), B, J'(1), \ldots, J'(n)) 
\quad \mapsto \quad PB \oplus PP^T A
\]
## Comparison between Category theory and Associative array

<table>
<thead>
<tr>
<th></th>
<th>Category theory</th>
<th>Associative array</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstraction level</strong></td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td><strong>Operation</strong></td>
<td>No concrete definition</td>
<td>Linear algebra operation</td>
</tr>
<tr>
<td><strong>Extensibility</strong></td>
<td>High, generalized to more data types</td>
<td>Focus on relation and graph</td>
</tr>
</tbody>
</table>
References on theoretical foundation (1)

**Category theory**

1. Multi-Model Database Management Systems-a Look Forward
ZH Liu, J Lu, D Gawlick, H Helskyaho, G Pogossiant, Z Wu, VLDB workshop Poly 2018


Associative array

Outline

- Motivation and multiple model examples (30')
- Theoretical foundations (30')
- Multi-model data storage (25')
- Questions and discussion (5')

Session break

- Multi-model data query languages (10')
- Multi-model query processing (10')
- Overview on tightly integrated polystores (20')
- Query processing in tightly integrated polystores (15')
- Advanced aspects of tightly integrated polystores (15')
- Comparison of multi-model databases and tightly integrated polystores (5')
- Open problems and challenges (10')
- Questions and discussion (5')
Multi-model data storage
Classification

- Basic approach: on the basis of original (or core) data model

<table>
<thead>
<tr>
<th>Original Type</th>
<th>Representatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational</td>
<td>PostgreSQL, SQL Server, IBM DB2, Oracle DB, Oracle MySQL, Sinew</td>
</tr>
<tr>
<td>Column</td>
<td>Cassandra, CrateDB, DynamoDB, HPE Vertica</td>
</tr>
<tr>
<td>Key/value</td>
<td>Riak, c-treeACE, Oracle NoSQL DB</td>
</tr>
<tr>
<td>Document</td>
<td>ArangoDB, Couchbase, MongoDB, Cosmos DB, MarkLogic</td>
</tr>
<tr>
<td>Graph</td>
<td>OrientDB</td>
</tr>
<tr>
<td>Object</td>
<td>Caché</td>
</tr>
<tr>
<td>Other</td>
<td>Not yet multi-model – NuoDB, Redis, Aerospike</td>
</tr>
<tr>
<td></td>
<td>Multi-use-case – SAP HANA DB, Octopus DB</td>
</tr>
</tbody>
</table>
Timeline

- When a particular system became multi-model
  - Original data format (model) was extended
  - First released directly as a multi-model DBMS
Extension towards Multiple Models

Types of strategies:

1. Adoption of a **completely new storage** strategy suitable for the new data model(s)
   - e.g., XML-enabled databases

2. **Extension of the original storage strategy** for the purpose of the new data model(s)
   - e.g., ArangoDB - special edge collections bear information about edges in a graph

3. Creating of a **new interface** for the original storage strategy
   - e.g., MarkLogic - stores JSON data in the same way as XML data

4. **No change** in the original storage strategy
   - Storage and processing of data formats simpler than the original one
<table>
<thead>
<tr>
<th>Approach</th>
<th>DBMS</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>New storage strategy</td>
<td>PostgreSQL</td>
<td>relational</td>
</tr>
<tr>
<td></td>
<td>SQL server</td>
<td>relational</td>
</tr>
<tr>
<td></td>
<td>IBM DB2</td>
<td>relational</td>
</tr>
<tr>
<td></td>
<td>Oracle DB</td>
<td>relational</td>
</tr>
<tr>
<td></td>
<td>Cassandra</td>
<td>column</td>
</tr>
<tr>
<td></td>
<td>CrateDB</td>
<td>column</td>
</tr>
<tr>
<td></td>
<td>DynamoDB</td>
<td>column</td>
</tr>
<tr>
<td></td>
<td>Riak</td>
<td>key/value</td>
</tr>
<tr>
<td></td>
<td>Cosmos DB</td>
<td>document</td>
</tr>
<tr>
<td>Extension of the original storage strategy</td>
<td>MySQL</td>
<td>relational</td>
</tr>
<tr>
<td></td>
<td>HPE Vertica</td>
<td>column</td>
</tr>
<tr>
<td></td>
<td>ArangoDB</td>
<td>document</td>
</tr>
<tr>
<td></td>
<td>MongoDB</td>
<td>document</td>
</tr>
<tr>
<td></td>
<td>OrientDB</td>
<td>graph</td>
</tr>
<tr>
<td></td>
<td>Caché</td>
<td>object</td>
</tr>
<tr>
<td>New interface for the original storage strategy</td>
<td>Sinew</td>
<td>relational</td>
</tr>
<tr>
<td></td>
<td>c-treeACE</td>
<td>key/value</td>
</tr>
<tr>
<td></td>
<td>Oracle NoSQL Database</td>
<td>key/value</td>
</tr>
<tr>
<td></td>
<td>Couchbase</td>
<td>document</td>
</tr>
<tr>
<td></td>
<td>MarkLogic</td>
<td>document</td>
</tr>
</tbody>
</table>
## Overview of Supported Data Models

<table>
<thead>
<tr>
<th>Type</th>
<th>DBMS</th>
<th>Relational</th>
<th>Column</th>
<th>Key/value</th>
<th>Document (JSON)</th>
<th>XML</th>
<th>Graph</th>
<th>Nested data/UDT/objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational</td>
<td>PostgreSQL</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>SQL Server</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>IBM DB2</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Oracle DB</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Oracle MySQL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Sinew</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Column</td>
<td>Cassandra</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CrateDB</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DynamoDB</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HPE Vertica</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Key/value</td>
<td>Riak</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c-treeACE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Oracle NoSQL DB</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Document</td>
<td>ArangoDB</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Couchbase</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>MongoDB</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Cosmos DB</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>MarkLogic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Graph</td>
<td>OrientDB</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Object</td>
<td>Caché</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Relational Stores

- **Representatives:** PostgreSQL, SQL Server, IBM DB2, Oracle DB, Oracle MySQL, Sinew
- **Biggest set of multi-model databases**
  - The most popular type of databases
  - SQL has been extended towards other data formats (e.g., SQL/XML)
  - Simplicity and universality of the relational model
CREATE TABLE customer (
    id INTEGER PRIMARY KEY,
    name VARCHAR(50),
    address VARCHAR(50),
    orders JSONB
);

INSERT INTO customer
VALUES (1, 'Mary', 'Prague',
        
        {"Order_no":"0c6df508",
         "Orderlines": [
            {
                "Product_no":"2724f", "Product_Name":"Toy", "Price":66,
                {
                "Product_no":"3424g", "Product_Name":"Book", "Price":40}
            }
        ]
        );

INSERT INTO customer
VALUES (2, 'John', 'Helsinki',
        
        {"Order_no":"0c6df511",
         "Orderlines": [
            { "Product_no":"2454f", "Product_Name":"Computer", "Price":34 }]
        );
SELECT json_build_object('id',id,'name',name,'orders',orders)
FROM customer;

SELECT jsonb_each(orders) FROM customer;

SELECT jsonb_object_keys(orders) FROM customer;
Column Stores

- **Representatives:** Cassandra, CrateDB, DynamoDB, HPE Vertica

- **Two meanings:**
  - Column-oriented (columnar, column) DBMS stores data tables as columns rather than rows
    - Not necessarily NoSQL
  - Column-family (wide-column) DBMS = a NoSQL database which supports tables having distinct numbers and types of columns
    - Underlying storage strategy is arbitrary
create keyspace myspace
WITH REPLACEMENT = { 'class' : 'SimpleStrategy', 'replication_factor' : 3 };

CREATE TYPE myspace.orderline ( product_no text, product_name text, price float );

CREATE TYPE myspace.myorder ( order_no text, orderlines list<frozen <orderline>> );

CREATE TABLE myspace.customer ( id INT PRIMARY KEY, name text, address text, orders list<frozen <myorder>> );
INSERT INTO myspace.customer JSON
' {"id":1,
  "name":"Mary",
  "address":"Prague",
  "orders" : [
  { "order_no":"0c6df508",
    "orderlines":[
    { "product_no" : "2724f",
      "product_name" : "Toy",
      "price" : 66 }
    , { "product_no" : "3424g",
        "product_name" : "Book",
        "price" : 40 } ] } ] }
'';

INSERT INTO myspace.customer JSON
' {"id":2,
  "name":"John",
  "address":"Helsinki",
  "orders" : [
  { "order_no":"0c6df511",
    "orderlines":[
    { "product_no" : "2454f",
      "product_name" : "Computer",
      "price" : 34 } ] } ] }
'';
CREATE TABLE myspace.users (
    id text PRIMARY KEY,
    age int,
    country text
);

INSERT INTO myspace.users (id, age, state)
VALUES ('Irena', 37, 'CZ');

SELECT JSON * FROM myspace.users;
[json]
-------------------------------------------
{"id": "Irena", "age": 37, "country": "CZ"}
Key/Value Stores

- Representatives: Riak, c-treeACE, Oracle NoSQL DB

- The simplest type of NoSQL database
  - Get / put / delete + key
  - Often extended with more advanced features

- Multi-model extensions:
  - More complex indices over the value part + new APIs (e.g., JSON, SQL, ...)

---

<table>
<thead>
<tr>
<th>Type</th>
<th>DBMS</th>
<th>Relational</th>
<th>Column</th>
<th>Key/Value</th>
<th>Document (JSON)</th>
<th>XML</th>
<th>Graph</th>
<th>Nested data/UDT/objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key/value</td>
<td>Riak</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c-treeACE</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oracle NoSQL DB</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
create table Customers (
    id integer,
    name string,
    address string,
    orders array (record (order_no string, orderlines array (record (product_no string, product_name string, price integer ))))
),
primary key (id)
);

import -table Customers -file customer.json

customer.json:
{
    "id":1,
    "name":"Mary",
    "address":"Prague",
    "orders" : [
        { "order_no":"0c6df508",
        "orderlines":[
            { "product_no" : "2724f",
            "product_name" : "Toy",
            "price" : 66 },
            { "product_no" : "3424g",
            "product_name"  :"Book",
            "price" : 40 } ] } ]
}
{
    "id":2,
    "name":"John",
    "address":"Helsinki",
    "orders" : [
        {"order_no":"0c6df511",
        "orderlines":[
            { "product_no" : "2454f",
            "product_name" : "Computer",
            "price" : 34 } ] } ]
}
```
sql-> select * from Customers
  -> ;

| id | name     | address   | orders         | orderlines
|----|----------|-----------|----------------|----------------
| 2  | John     | Helsinki | order_no       | 0c6df511
|    |          |           | orderlines     |                |
|    |          |           | product_no     | 2454f          |
|    |          |           | product_name   | Computer       |
|    |          |           | price          | 34             |
| 1  | Mary     | Prague    | order_no       | 0c6df508
|    |          |           | orderlines     |                |
|    |          |           | product_no     | 2724f          |
|    |          |           | product_name   | Toy            |
|    |          |           | price          | 66             |
|    |          |           | product_no     | 3424g          |
|    |          |           | product_name   | Book           |
|    |          |           | price          | 40             |
```
Document Stores

- Representatives: ArangoDB, Couchbase, MongoDB, Cosmos DB, MarkLogic
- Distinct strategies:
  - ArangoDB: special edge collection
  - MarkLogic: stores JSON data as XML
MarkLogic

```javascript
function declareUpdate() {
  xdmp.documentInsert("/myJSON1.json", {
    "name": "Oliver",
    "scores": [88, 67, 73],
    "isActive": true,
    "affiliation": null
  });
}
```

```
XQuery:
xdmp:document-insert("/myXML1.xml",
  <product no="3424g">
    <name>The King's Speech</name>
    <author>Mark Logue</author>
    <author>Peter Conradi</author>
  </product>
);
Graph Stores

- Representatives: OrientDB
- Based on an object database = native support for multiple models
  - Element of storage = record = document / BLOB / vertex / edge
- Classes – define records
- Classes can have relationships:
  - Referenced – stored similarly to storing pointers between two objects in memory
  - Embedded – stored within the record that embed

<table>
<thead>
<tr>
<th>Type</th>
<th>DBMS</th>
<th>Relational</th>
<th>Column</th>
<th>Key/Value</th>
<th>Document (JSON)</th>
<th>XML</th>
<th>Graph</th>
<th>Nested data/UDT/object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph</td>
<td>OrientDB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Frank

```
{  "@grid": "#12:302",  "@class": "Customer",  "name": "Frank",  "surname": "Raggio",  "phone": ["+44 123456", "+44 321543"],  "details": {    "city": "London",    "tags": ["millennial"]  }}
```

Order

Couchbase

- OrientDB
- MySQL
- ArangoDB
- DynamoDB
- Cassandra
- Couchbase
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (JSON)
- HPE Vertica
- Oracle NoSQL DB
- Oracle DB (JSON)
- PostgreSQL (JSON)
- Oracle DB (J
CREATE CLASS orderline EXTENDS V
CREATE PROPERTY orderline.product_no STRING
CREATE PROPERTY orderline.product_name STRING
CREATE PROPERTY orderline.price FLOAT

CREATE CLASS order EXTENDS V
CREATE PROPERTY order.order_no STRING
CREATE PROPERTY order.orderlines EMBEDDEDLIST orderline

CREATE CLASS customer EXTENDS V
CREATE PROPERTY customer.id INTEGER
CREATE PROPERTY customer.name STRING
CREATE PROPERTY customer.address STRING

CREATE CLASS orders EXTENDS E

CREATE CLASS knows EXTENDS E
CREATE VERTEX order CONTENT {
    "order_no": "0c6df508",
    "orderlines": [
        {
            "@type": "d",
            "@class": "orderline",
            "product_no": "2724f",
            "product_name": "Toy",
            "price": 66
        },
        {
            "@type": "d",
            "@class": "orderline",
            "product_no": "3424g",
            "product_name": "Book",
            "price": 40
        }
    ]
}

CREATE VERTEX order CONTENT {
    "order_no": "0c6df511",
    "orderlines": [
        {
            "@type": "d",
            "@class": "orderline",
            "product_no": "2454f",
            "product_name": "Computer",
            "price": 34
        }
    ]
}

CREATE VERTEX customer CONTENT {
    "id": 1,
    "name": "Mary",
    "address": "Prague"
}

CREATE VERTEX customer CONTENT {
    "id": 2,
    "name": "John",
    "address": "Helsinki"
}
CREATE EDGE orders FROM 
(SELECT FROM customer WHERE name = "Mary")
TO
(SELECT FROM order WHERE order_no = "0c6df508")
CREATE EDGE orders FROM
(SELECT FROM customer WHERE name = "John")
TO
(SELECT FROM order WHERE order_no = "0c6df511")
CREATE EDGE knows FROM
(SELECT FROM customer WHERE name = "Mary")
TO
(SELECT FROM customer WHERE name = "John")
Question?
Break
Outline

- Motivation and multiple model examples (30')
- Theoretical foundations (30')
- Multi-model data storage (25')
- Questions and discussion (5')

Session break

- Multi-model data query languages (10')
- Multi-model query processing (10')
- Overview on tightly integrated polystores (20')
- Query processing in tightly integrated polystores (15')
- Advanced aspects of tightly integrated polystores (15')
- Comparison of multi-model databases and tightly integrated polystores (5')
- Open problems and challenges (10')
- Questions and discussion (5')
Multi-model data query languages
Multi-model Query Languages

1. Simple API
   ○ Store, retrieve, delete data
     ■ Typically key/value, but also other use cases
   ○ DynamoDB – simple data access + querying over indices using comparison operators

2. SQL Extensions and SQL-Like Languages
   ○ Most common
   ○ In most types of systems (relational, column, document, …)
<table>
<thead>
<tr>
<th>Type</th>
<th>DBMS</th>
<th>SQL extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational</td>
<td>PostgreSQL</td>
<td>Getting an array element by index, an object field by key, an object at a specified path, containment of values/paths, top-level key-existence, deleting a key-value pair / a string element / an array element with specified index / a field / an element with specified path, ...</td>
</tr>
<tr>
<td>SQL Server</td>
<td></td>
<td>JSON: export relational data in the JSON format, test JSON format of a text value, JavaScript-like path queries</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLXML: SQL view of XML data + XML view of SQL relations</td>
</tr>
<tr>
<td></td>
<td>IBM DB2</td>
<td>SQL/XML + embedding SQL queries to XQuery expressions</td>
</tr>
<tr>
<td></td>
<td>Oracle DB</td>
<td>SQL/XML + JSON extensions (JSON_VALUE, JSON_QUERY, JSON_EXISTS, ...</td>
</tr>
<tr>
<td>Document</td>
<td>Couchbase</td>
<td>Clauses SELECT, FROM (multiple buckets), ... for JSON</td>
</tr>
<tr>
<td></td>
<td>Cosmos DB</td>
<td>Clauses SELECT, FROM (with inner join), WHERE and ORDER BY for JSON</td>
</tr>
<tr>
<td></td>
<td>ArangoDB</td>
<td>key/value: insert, look-up, update document: simple QBE, complex joins, functions, ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>graph: traversals, shortest path searches</td>
</tr>
<tr>
<td>Key/value</td>
<td>Oracle NoSQL DB</td>
<td>SQL-like, extended for nested data structures</td>
</tr>
<tr>
<td></td>
<td>c-treeACE</td>
<td>Simple SQL-like language</td>
</tr>
<tr>
<td>Column</td>
<td>Cassandra</td>
<td>SELECT, FROM, WHERE, ORDER BY, LIMIT with limitations</td>
</tr>
<tr>
<td></td>
<td>CrateDB</td>
<td>Standard ANSI SQL 92 + nested JSON attributes</td>
</tr>
<tr>
<td>Graph</td>
<td>OrientDB</td>
<td>Classical joins not supported, the links are simply navigated using dot notation; main SQL clauses + nested queries</td>
</tr>
<tr>
<td>Object</td>
<td>Caché</td>
<td>SQL + object extensions (e.g. object references instead of joins)</td>
</tr>
</tbody>
</table>
SELECT name,
    orders->>'Order_no' AS Order_no,
    orders#>'{Orderlines,1}'->>'Product_Name' AS Product_Name
FROM customer
WHERE orders->>'Order_no' <> '0c6df511';
```
sql-> select * from Customers
  -> ;

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>address</th>
<th>order_no</th>
<th>orders</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>John</td>
<td>Helsinki</td>
<td>0c6df511</td>
<td>orderlines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>product_no</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>product_name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>price</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>Mary</th>
<th>Prague</th>
<th>0c6df508</th>
<th>orderlines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>product_no</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>product_name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>price</td>
</tr>
</tbody>
</table>

|    |       |          |          | product_no | 3424g |
|    |       |          |          | product_name | Book |
|    |       |          |          | price | 40 |
```
sql-> SELECT c.name, c.orders.order_no, c.orders.orderlines[0].product_name
       -> FROM customers c
       -> WHERE c.orders.orderlines[0].price > 50;

+-----------------+-------------------+-----------------------+
| name            | order_no          | product_name          |
+-----------------+-------------------+-----------------------+
| Mary            | 0c6df508          | Toy                   |
+-----------------+-------------------+-----------------------+

sql-> SELECT c.name, c.orders.order_no,
       -> [c.orders.orderlines[$element.price > 35]]
       -> FROM customers c;

+-----------------+-------------------+-----------------------+-----------------+-------------------+
| name            | order_no          | Column_3              | product_no      | 2724f             |
| Mary            | 0c6df508          | product_name          | Toy             |                  |
|                 |                   | price                 | 66              |                  |
|                 |                   | product_no            | 3424g           |                  |
|                 |                   | product_name          | Book            |                  |
|                 |                   | price                 | 40              |                  |
+-----------------+-------------------+-----------------------+-----------------+-------------------+
| John            | 0c6df511          |                       |                 |                  |
+-----------------+-------------------+-----------------------+-----------------+-------------------+
Multi-model Query Languages

3. SPARQL Query Extensions
   - e.g., IBM DB2 - SPARQL 1.0 + subset of features from SPARQL 1.1
     - SELECT, GROUP BY, HAVING, SUM, MAX, …
     - Probably no extension for relational data
       - But: RDF triples are stored in a table = SQL queries can be used over them too

4. XML Query Extensions
   - MarkLogic – JSON can be accessed using XPath
     - Tree representation like for XML
     - Can be called from XQuery and JavaScript

5. Full-text Search
   - In general quite common
   - e.g., Riak – Solr index + operations
     - Wildcards, proximity search, range search, Boolean operators, grouping, …
JavaScript:
declareUpdate();
xdmp.documentInsert("/myJSON1.json",
{
    "Order_no":"0c6df508",
    "Orderlines":[
        {
            "Product_no":"2724f",
            "Product_Name":"Toy",
            "Price":66
        },
        {
            "Product_no":"3424g",
            "Product_Name":"Book",
            "Price":40
        }
    ]
});

XQuery:
xdmp:document-insert("/myXML1.xml",
<product no="3424g">
    <name>The King's Speech</name>
    <author>Mark Logue</author>
    <author>Peter Conradi</author>
</product>
);

XQuery:
let $product := fn:doc("/myXML1.xml")/product

let $order := fn:doc("/myJSON1.json")
    [Orderlines/Product_no = $product/@no]
return $order/Order_no
Result: 0c6df508
Outline

● Motivation and multiple model examples (30’)
● Theoretical foundations (30’)
● Multi-model data storage (25’)
● Questions and discussion (5’)

Session break

● Multi-model data query languages (10’)
● Multi-model query processing (10’)
● Overview on tightly integrated polystores (20’)
● Query processing in tightly integrated polystores (15’)
● Advanced aspects of tightly integrated polystores (15’)
● Comparison of multi-model databases and tightly integrated polystores (5’)
● Open problems and challenges (10’)
● Questions and discussion (5’)

Multi-model query processing
Query Processing Approaches

● Depend highly on the way the system was extended
  ○ No change
  ○ New interface
    ■ e.g., MarkLogic
  ○ Extension of the original storage strategy
    ■ e.g. ArangoDB
  ○ A completely new storage strategy
    ■ e.g. Oracle native support for XML

● General tendencies:
  ○ Exploit the existing storage strategies as much as possible
  ○ Exploit the verified approaches to query optimization

changes in the query processing approaches
JavaScript:
```
declareUpdate();
xdmp.documentInsert("/myJSON1.json",
{
    "Order_no": "0c6df508",
    "Orderlines": [
        {
            "Product_no": "2724f",
            "Product_Name": "Toy",
            "Price": 66
        },
        {
            "Product_no": "3424g",
            "Product_Name": "Book",
            "Price": 40
        }
    ]
});
```

XQuery:
```
xdmp:document-insert("/myXML1.xml",
<product no="3424g">
    <name>The King's Speech</name>
    <author>Mark Logue</author>
    <author>Peter Conradi</author>
</product>
);
```

XQuery:
```
let $product := fn:doc("/myXML1.xml")/product
let $order := fn:doc("/myJSON1.json")
    [Orderlines/Product_no = $product/@no]
return $order/Order_no
Result: 0c6df508
```
MarkLogic Multiple Models

- Indexes both XML and JSON data in the same way
- Schema-less data
- Universal index - optimized to allow text, structure and value searches to be combined into
  - Word indexing
  - Phrase indexing
  - Relationship indexing
  - Value indexing
- Other user-defined indexes
  - Range indexing
  - Word lexicons
  - Reverse indexing
  - Triple index
### Social network graph

- **Marry (1)**
  - Friend: **William (3)**
  - Friend: **John (2)**

### Key/value pairs

- 
  - `(Customer_ID, Order_no)`: 
    - "1" -> "34e5e759"
    - "2" -> "0c6df508"

### Order JSON document

```
{"Order_no": "0c6df508",
 "Orderlines": [
  {
    "Product_no": "2724f",
    "Product_Name": "Toy",
    "Price": 66
  },
  {
    "Product_no": "3424g",
    "Product_Name": "Book",
    "Price": 40
  }
]}
```

### Relation Customers

<table>
<thead>
<tr>
<th>Customer_ID</th>
<th>Name</th>
<th>Credit_limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mary</td>
<td>5,000</td>
</tr>
<tr>
<td>2</td>
<td>John</td>
<td>3,000</td>
</tr>
<tr>
<td>3</td>
<td>William</td>
<td>2,000</td>
</tr>
</tbody>
</table>
LET CustomerIDs = ( 
  FOR Customer IN Customers 
  FILTER Customer.CreditLimit > 3000 
  RETURN Customer.id )

LET FriendIDs = ( 
  FOR CustomerID IN CustomerIDs 
  FOR Friend IN 1..1 OUTBOUND CustomerID Knows 
  RETURN Friend.id )

FOR Friend IN FriendIDs 
  FOR Order IN 1..1 OUTBOUND Friend Customer2Order 
  RETURN Order.orderlines[*].Product_no

Return all products which are ordered by a friend of a customer whose credit limit is over 3000.
ArangoDB Multiple Models

● Supported models:
  ○ Document - original
  ○ Key/value - special type of document without complex value part
  ○ Tables - special type of document with regular structure
  ○ Graph - relations between documents
    ■ Edge collection – two special attributes _from and _to

● So we still need to efficiently process queries over documents

● Indexes
  ○ Primary = hash index for document keys
  ○ Edge = hash index, which stores the union of all _from and _to attributes
    ■ For equality look-ups
  ○ User-defined - (non-)unique hash/skiplist index, (non-)unique sparse hash/skiplist index, geo, fulltext, ...
Query Optimization Strategies

- B-tree/B+-tree index - the most common approach
  - Typically in relational databases
- Native XML index - support of XML data
  - Typically an ORDPATH-based approach
- Hashing - can be used almost universally
- ...
- But: still no universally acknowledged optimal or sub-optimal approach
  - Approaches are closely related to the way the system was extended
Outline

● Motivation and multiple model examples (30')
● Theoretical foundations (30')
● Multi-model data storage (25')
● Questions and discussion (5')

Session break

● Multi-model data query languages (10')
● Multi-model query processing (10')
● **Overview on tightly integrated polystores (20')**
● Query processing in tightly integrated polystores (15')
● Advanced aspects of tightly integrated polystores (15')
● Comparison of multi-model databases and tightly integrated polystores (5')
● Open problems and challenges (10')
● **Questions and discussion (5')**
Overview on tightly integrated polystores
No one size fits all…

- Heterogenous analytics: data processing frameworks (MR, Spark, Flink), NoSQL
- ETL is very expensive towards a single model (may degrade performance), adapts poorly to changes in data / application requirements

Polystore idea: package together multiple query engines: union (federation) of different specialized stores, each with distinct (native) data model, internal capabilities, language, and semantics → **Holy grail: platform agnostic data analytics**

- Use the right store for (parts of) each specialized scenario
- Possibly rely on middleware layer to integrate data from different sources
- Read-only queries as distributed transactions over different data stores is hard!
Dimensions of polystores *

- Heterogeneity – different data models / query models, semantic expressiveness / query engines
- Autonomy – association, execution, evolution
- Transparency – location (data may even span multiple storage engines), transformation / migration
- Flexibility – schema, interfaces, architecture
- Optimality – federated plans, data placement

Tightly integrated polystores (TIPs)

- Heterogeneity moderate
- Autonomy low
- Transparency high
- Flexibility low
- Optimality high
- Semantic expressiveness high
TIPs

- Trade autonomy for efficient querying of diverse kinds of data for BD analytics
  - data stores can only be accessed through the multi-store system (slaves)
  - less uncertainty with extended control over the various stores
  - stores accessed directly through their local language
- Query processor directly uses the local language and local interfaces
- Efficient / adaptive data movement across data stores
- Number of data stores that can be interfaced is typically limited
- Extensibility? Good to have…

Arguably the closest we can get to multi-model DBs, while having several native stores “under the hood”.
Loosely integrated polystores

Reminiscent of multidatabase systems, follow mediator-wrapper architecture (one wrapper per datastore), one global common language

- Notable examples: BigIntegrator, Forward/SQL++, QoX
- Data mediation SQL engines: Apache Drill, Spark SQL, SQL++ allow different sources to be plugged in by wrappers, then queried via SQL

General approach

- Split a query into subqueries (per datastore, still in common language)
- Send to wrapper, translate, get results, translate to common format, integrate
Hybrid polystores

Rely on tight coupling for some stores, loose coupling for others, following the mediator-wrapper architecture, but the query processor can also directly access some data stores

- Notable examples: BigDawg (next), SparkSQL, CloudMdsQL
BigDawg – Big Data Analytics Working Group*

- One key abstraction: island of information, a collection of data stores accessed with a single query language
- BigDawg relies on a variety of data islands (relational, array, NoSQL, streaming, etc)
- No common data model, query language / processor (each island has its own)
- Wrappers (shims) mapping the island query to the native one
- CAST: explicit operators for moving intermediate datasets between islands
- Subqueries for multi-island query processing

* https://bigdawg.mit.edu/
Historical perspective

Multi-database systems (federated systems, data integration systems)
- mediator-wrapper architecture, declarative SQL-like language, single unified global schema (GAV, LAV)
- key principle: query is sent to store that owns the data
- focus on data integration

The reference federated databases: Garlic, Tsimmis
- even multi-model settings, but the non-relational stores did not support their own declarative query language (being wrapped to provide an SQL API)
- no cross-model rewriting

Polystores:
- higher expectations in terms of data heterogeneity
- allow the direct exploitation of the datasets in their native language (but not only)
Another classification for polystores / multistores*

- Federated systems: collection of homogeneous data stores and features a single standard query interface
- Polyglot systems: collection of homogeneous data stores and exposes multiple query interfaces to the users
- Multistore systems: data across heterogeneous data stores, while supporting a single query interface
- Polystore systems: query processing across heterogeneous data stores and supports multiple query interfaces

Scenarios for polystores*

- Platform independence
- Data analysis spanning stores (polystore)
- Query acceleration / opportunistic cross-platform
- Mandatory cross-platform

In summary - goals of TIPs

- Focus on efficiency and transparency
- Exploit mature, focused technologies, good fits for different workloads
- Integrated, transparent access to data stores through one or more query languages (semantic expressiveness)
- Exploit the full expressive power of the native query languages
- Ease of use / develop apps
In summary - goals of TIPs (cont’d)

- Cross-model data migration, automated scheduling, self tuning (transparent)
- Cross-platform / multi-model query planning and optimizer
  - automatic query reformulation, inter-platform parallelism, …
- Potential goal: internally, unified storage abstraction
  - cross-model view (internal) over the native data
Main TIPs aspects discussed

- Architecture
- Data models / storage
- Query languages
- Query processing

Systems: HadoopBD, Polybase, Estocada/Tatooine, Odyssey/MISO, Myria, RHEEM
HadoopDB* - introduction

Main idea:

- Query RDBMS from Hadoop
- use MR as communication layer

Schema: GAV

Queries: SQL-like system (HiveQL)

Objective: the best of parallel DBMS and MR systems, gets efficiency of PDBMS and scalability, fault-tolerance of MR
- Extends HIVE (Facebook) to push down operations into single node DBMS

* http://dslam.cs.umd.edu/hadoopdb/hadoopdb.html
HadoopDB - introduction (cont’d)

- Multiple single-node RDBMs (PostgreSQL, VectorWise) coupled with HDFS/MR, deployed in a shared-nothing cluster
- Extensions to Hadoop: DB connector, catalog, data loader
- SQL-MR-SQL planner: extends HIVE, HiveQL → MR → SQL
- Data is partitioned both in RDMS tables and in HDFS files
HadoopDB - big picture
HadoopDB data and query model

- Raw data (text / binary), transformed into key-value pairs for Map tasks
  - Data globally repartitioned on a given key
  - Building and bulk-loading data chunks in the single-node DBs
- Relational data (column store or row store) → rows also hash partitioned across BD instances

Queries expressed as SQL (front end, extends HIVE)
- translated into MR, work pushed to single node DBMSs
Polybase* - introduction

Main idea:

- querying Hadoop (unstructured data) from RDBMS (structured)
- SQL Server Parallel Data Warehouse (shared nothing parallel) + Hadoop

Schema: GAV

Queries: SQL queries and distributed SQL execution plans

Objective: data in Hadoop (people just don’t see the value of clean, schema, load… or are more comfortable writing procedural code)

- Minimize data imported to PDW, maximize MR processing capability

Polybase - introduction (cont’d)

- HDFS data can be imported / exported to / from SQL Server PDW
- HDFS referenced as « external tables », manipulated together with PDW native tables
- Takes advantage of PDW’s data movement service (DMS), extended with HDFS bridge

(a) PDW query in, results out  
(b) PDW query in, results stored in HDFS
Polybase data and query model

- Raw data files (text / binary) - unstructured data with relational view
- Relational data - structured
- Queries expressed as SQL over relational tables (including external ones)
  - Translates SQL operators into MR jobs for data in HDFS

```sql
CREATE EXTERNAL TABLE hdfsCustomer
  ( c_custkey bigint not null,
    c_name varchar(25) not null,
    c_address varchar(40) not null,
    c_nationkey integer not null,
    c_phone char(15) not null,
    c_acctbal decimal(15,2) not null,
    c_mktsegment char(10) not null,
    c_comment varchar(117) not null
  ) WITH (LOCATION='/tpch1gb/customer.tbl',
         FORMAT_OPTIONS (EXTERNAL_CLUSTER = GSL_CLUSTER,
                         EXTERNAL_FILEFORMAT = TEXT_FORMAT));
```
Estocada* - introduction

Main idea: self-tuning platform supporting natively various models

Schema: LAV

Queries: Access to each dataset in its native format
  - no common query language / data model on top

Objectives: allow any data model, at both application and view level
  - fragment based store, transparent to users
    - automatically distribute / partition the data into fragments
  - although accessed natively, data internally may reside in different formats
    - pivot language: relational with prominent use of constraints

Estocada - big picture
Estocada - data and query model

- (Nested) relational data, NoSQL (graphs, key-value, document)
- Queries expressed natively (e.g., over JSON data, below), translated into pivot language → relational algebra
**Tatooine** - introduction

Main idea: use ontologies to mediate relational and non-relational sources

- RDF model as “glue” between all other models

Schema: GAV

Queries: Conjunctive Mixed Queries (CMQ) - variation over the SPARQL subset of conjunctive queries (a.k.a. Basic Graph Pattern Queries - BGPs)

Objectives: lightweight integration over multiple native stores (mixed data instance), with focus on querying with a unified view

- a specific architecture and usage scenario for data journalism
- custom (application dependent) RDF graph, including ontology / triples, acting as bridge between different stores, based on common / repeated values (URIs)

Tatooine - big picture

Data sources
- JSON sources (SOLR, Postgres...)
- Relational sources (RDBMS)
- RDF sources (Jena, Corese...)
- Streams (Twitter, RSS)

Data processing
- Classification, ...
- Custom processing

Querying and analysis
- Dataviz modules
- Tatooine query engine
- Full-text search
- SQL Indexing
- SPARQL Reasoning
- Subscribe
- Query
Odyssey / Miso* - introduction

Main idea: self-tuning polystore on different analytic engines (parallel OLAP, Hadoop)

- enables storing and querying in HDFS and relational stores, using opportunistic materialized views

Schema: LAV

Queries: SQL-like (HiveQL) posed on HDFS, RDBMS used as query accelerator

Objectives: focus on time-to-insight / evolutionary analytics

- which dataset should we move, where, when → method for tuning the physical design (MISO), decide in which data store the data should reside

* LeFevre et al. “MISO: souping up big data query processing with a multistore system”. SIGMOD 2014
Odyssey / Miso - introduction (cont’d)

- Insight: single query optimization over multi-stores brings limited benefits; workload optimization instead
- Claim: physical design tuning is key
- Continuously monitors the workloads, online analysis to decide which views to materialize (share computation across queries)
Odyssey / Miso - big picture
Odyssey - data and query model

- Structured (relational) and unstructured data (large log files, text-based data stored as flat HDFS files)

- HiveQL queries: declarative language on top of MR
  - relational operators and arbitrary code (UDFs)
  - UDFs executed as MR jobs
Myria* - introduction

Main idea: A federated data analytics system, over data held by multiple backend systems (including MyriaX, SciDB, PostgreSQL, RDF, Spark key-value store)

Schema: LAV

Queries: MyriaL - a hybrid declarative / imperative language (relational query language with imperative extensions) (or Python)

Objectives: “relational at the core approach”, focus on efficiency and usability, delivering the performance of specialized systems with the convenience of general purpose systems

- hides the data model differences between the various backends

* http://myria.cs.washington.edu/
Myria - big picture

- MyriaX (a parallel shared nothing DBMS) - query execution engine
- PipeGen: automatic data migration between stores, in support of query plans across engine boundaries
- RACO: Relational Algebra Compiler (locality-aware, rule based)
Myria - data and query model

- Relations, arrays, graphs, key-value pairs → the relational data model is used for translation and optimization
  - Observation: fundamentally isomorphic
- Queries expressed as SQL with imperative statements (similar to PL/SQL):
  - Relational semantics defined for operators of non-relational systems
  - Rules to translate such operators properly

```plaintext
1   E = scan(Graph); -- Graph(x, y) is an edge table
2   V = select distinct x from E;
3   CC = [from V emit x as node_id, x as comp_id];
4   do
5       newCC = CC + [from E, CC where E.x = CC.node_id
6           emit E.y, CC.comp_id];
7       newCC = [from newCC emit
8           newCC.node_id, min(new_CC.comp_id) as comp_id];
9       delta = diff(CC, newCC);
10      CC = newCC;
11   while [from delta emit count(*) > 0];
12   components = [from CC emit CC.comp_id, count(CC.node_id)];
13   store(components, ConnectedComponents);
```
RHEEM* - introduction

Main idea: general purpose cross-platform data processing system (DBMS, MR, NoSQL) -- data natively resides on different storage platforms

Schema: LAV

Queries: logic of the app in imperative form

Objectives: Decouple applications from underlying platforms → multi-platform task execution and data storage independence

- platform independent task specification
- transparent multi-platform optimization & execution (cost-based / learned)
- data storage and data movement optimization
- data processing and storage abstraction for adaptability / extensibility

* http://da.qcri.org/rheem/
RHEEM - big picture
RHEEM - data and query model

- Data quanta abstraction (for database tuples, graph edges, full document content, etc)
- Procedural data-flow queries (Rheem plan)
  - Rheem Latin (based on Pig Latin grammar), Rheem Studio
  - Data-flow graph, vertices being platform agnostic operators
  - One or several data source

```java
import '/sgd/udfs.class' AS taggedPointCounter;
lines = load 'hdfs://myData.csv';
points = map lines -> {taggedPointCounter.parsePoints(lines)};
weights = load taggedPointCounter.createWeights();
final_weights = repeat 50 {
    sample_points = sample points -> {taggedPointCounter.getSample()}
        with broadcast weights;
    gradient = map sample_points ->
        {taggedPointCounter.computeGradient()};
    gradient_sum_count = reduce gradient -> {gradient.sumcount()};
    weights = map gradient_sum -> {gradient_sum_count.average()} with platform 'JavaStreams';
    store final_weights 'hdfs://output/sgd';
}
```
Outline

- Motivation and multiple model examples (30')
- Theoretical foundations (30')
- Multi-model data storage (25')
- Questions and discussion (5')

Session break

- Multi-model data query languages (10')
- Multi-model query processing (10')
- Overview on tightly integrated polystores (20')
  - Query processing in tightly integrated polystores (15')
  - Advanced aspects of tightly integrated polystores (15')
  - Comparison of multi-model databases and tightly integrated polystores (5')
  - Open problems and challenges (10')
  - Questions and discussion (5')
Query processing in TIPs
HadoopDB

Query processing: split MR/DB joins, referential partitioning, post-join aggregation
Query optimization: heuristics

Queries expressed as SQL (front end, extends HIVE), translated into MR, work pushed to single node DBMSs

- Query processing is simple: HiveQL query decomposed into QEP of relational operators, which are translated into MR jobs
- Leaf nodes are transformed into SQL to query the RDBMS instances
- Joins: easy if corresponding partitions collocated on same physical node
HadoopDB (cont’d) - SMS planner (extending Hive)

```
SELECT YEAR(date) AS year, SUM(price)
FROM sales
GROUP BY year
```

SMS Planner extensions, before execution:

- Retrieve data fields to determine partitioning keys
- Traverse DAG bottom-up (rule based SQL generator)
Polybase

- Query plans: search space with 2 parts
  - MR jobs
  - regular relational operators
- Cost-based query optimizer: decide when good to push SQL to HDFS (statistics on external tables)
  - selects and projects on external tables (by MR jobs)
  - joins of 2 external tables (only when both tables are stored in HDFS)
  - indexes built on HDFS-resident data, stored inside PDW → use as pre-filter, lazily updated
- Query processing: query splitting
Polybase example

```sql
SELECT count (*) from Customer
WHERE acctbal < 0
GROUP BY nationkey
```
Polybase example (cont’d)

(a) Optimized query plan

(b) Corresponding DSQL plan

(a) Alternative query plan

(b) Corresponding DSQL plan

Normal Agg

RETURN:
SELECT COUNT(*) FROM TEMP_1 WHERE ACCTBAL < 0 GROUP BY NATIONKEY

DMS:
Import hdfsCustomer into TEMP_1 distributed on NATIONKEY

CREATE:
CREATE TABLE TEMP_1(ACCTBAL, NATIONKEY) WITH DISTRIBUTION = NATIONKEY

Filter

Scan Customer

HDFS Shuffle

Final Agg

Partial Agg

Filter

Scan Customer

Hadoop Map Job

HADOOP:
Run Map job computing filter and partial count over Customer in HDFS. Results are stored in HDFSInter.

RETURN:
SELECT SUM(PartialCount) FROM TEMP_1 GROUP BY NATIONKEY

DMS:
Import HDFSInter into TEMP_1 distributed on NATIONKEY

CREATE:
CREATE TABLE TEMP_1(PartialCount, NATIONKEY) WITH DISTRIBUTION = NATIONKEY
Estocada

- Recall: fragment based store, automatically distributes / partition the data into fragments → each data partition described as a materialized view
- View-based query processing: with conjunctive queries + constraints
- Query optimization: cost based
- Query → logical QEP on possibly multiple data stores → physical QEP based on relational algebra
  - leafs being translated into queries accessing the stores natively
  - work divided between the native stores and Estocada’s own runtime engine
- Cross model / language storage advising (akin to automatic view selection)
MISO / Odyssey

- Optimization for entire workloads, using opportunistic materialized views
  - Shared intermediate results: opportunistic materialized views (useful if used repeatedly)
- Normalized cost-based optimization
  - Cost in HV
  - Cost in DW
  - Cost of transfer between stores
- Annotated (by views) Query Plans
  - Stores in which sub-expressions are computed depends on the multistore physical design (views)
Myria

- Relational algebra compiler (RACO) is the query optimizer and federated query executor
- MyriaX takes as input RACO query plans
- RACO uses rule-based optimization
  - default: each leaf assigned to where data resides
  - iterates bottom up adding data movement operators wherever needed
  - rewrite rules determine the platform that each operator should run on
Myria RACO optimizer

- Graph of operators (including cycles, for iterative processing)
  - including relational operators + iterative processing, flatmap, stateful apply
- Query plans are organized into fragments and are fully pipelined
- Efficient join query evaluation (for large tables)
- Data movement during federated execution: if a query spans engine boundaries, intermediate results must move across systems
  - via HDFS, in a common format (CVS), or
  - new interconnection operators for each pair of systems, or...
  - PipeGen: enables automatically optimized data transfer between arbitrary pairs of systems
RHEEM

- **Input:** query – logic of the app in imperative form
- **System ultimately decides on which platform to execute each task**

For each RHEEM operator list all the alternatives for the optimizer to choose from:
inflate RHEEM plan (each operator + all all its execution alternatives)

Three layer decoupled optimization

- **Logical operators** (application optimization layer)
- **Physical operators** (core optimization layer)
- **Execution operators** (platform optimization layer)
RHEEM execution plan

Cost-based optimizer outputs a RHEEM execution plan, also a data flow graph, but:

- vertices are platform-specific execution operators
- may include operators for data movement across platforms

Cost model:

- each execution operator has an estimated cost, based on resource usage and unit cost, data cardinality → user hints, learned from logs, progressive optimization
- data movement: planning and assessing for cost model optimization
  - channel conversion graph (CCG): space of possible communication steps
RHEEM - inflated execution plans

(a) RHEEM plan

(b) Execution plan

RHEEM operator
Spark execution operator
JavaStreams execution operator
Input/Output
UDF
Data flow
Broadcast data flow

TextFile Source
Map
parse
Collection Source
RepeatLoop
Collection Sink
ReduceBy
sum & count
Map
average

(a) Operator mappings

(b) Inflated operator with estimates

ReduceBy
assign
Map
assign
ReduceBy
assign
RepeatLoop
sum & count
Collection Source
ReduceBy
sum & account
Collection Sink
Map
average
ReduceBy
sum & account
Map
assign
ReduceBy
sum & account
GroupBy
output cardinality estimate
960, 1100
conf. 95%
cost estimate
[2.45e, 3.33e]
conf. 80%

(a) 1-to-1 mapping
(b) 1-to-n mapping
(c) n-to-1 mapping
(d) m-to-n mapping
Outline

- Motivation and multiple model examples (30')
- Theoretical foundations (30')
- Multi-model data storage (25')
- Questions and discussion (5')

Session break

- Multi-model data query languages (10')
- Multi-model query processing (10')
- Overview on tightly integrated polystores (20')
- Query processing in tightly integrated polystores (15')
  - Advanced aspects of tightly integrated polystores (15')
- Comparison of multi-model databases and tightly integrated polystores (5')
- Open problems and challenges (10')
- Questions and discussion (5')
Advanced Aspects of TIPs
Tuning (MISO example)

- Physical design: materialized views
  - View storage budget
  - View migration budget
- Reorganization phase (workload history)
- Computationally hard problem
  - Heuristic approach: variant of the knapsack problem
  - Additional complexity: 2 physical design pbs, each with 2 dimensions

**Multistore Design Problem.** Given an observed query stream, a multistore design $\mathcal{M} = <\mathcal{V}_h, \mathcal{V}_d>$, and a set of design constraints $B_h, B_d, B_t$ compute a new multistore design $\mathcal{M}^{\text{new}} = <\mathcal{V}_h^{\text{new}}, \mathcal{V}_d^{\text{new}}>$, where $\mathcal{V}_h^{\text{new}}, \mathcal{V}_d^{\text{new}}$ in $\mathcal{V}_h \cup \mathcal{V}_d$, that satisfies the constraints and minimizes future workload cost.
MISO tuner algorithm

Algorithm 1 MISO Tuner algorithm

1: function MISO_TUNE($\langle V_h, V_d \rangle, W, B_h, B_d, B_t$)
2: \hspace{1cm} $V \leftarrow V_h \cup V_d$
3: \hspace{1cm} $P \leftarrow \text{COMPUTEINTERACTINGSETS}(V)$
4: \hspace{1cm} $V_{cands} \leftarrow \text{SPARSIFFYSETS}(P)$
5: \hspace{1cm} $V_{d}^{\text{new}} \leftarrow \text{M-KNAPSACK}(V_{cands}, B_d, B_t)$
6: \hspace{1cm} $B_t^{\text{rem}} \leftarrow B_t - \sum_{v \in V_h \cap V_d^{\text{new}}} \text{sz}(v)$
7: \hspace{1cm} $V_{h}^{\text{new}} \leftarrow \text{M-KNAPSACK}(V_{cands} - V_d^{\text{new}}, B_h, B_t^{\text{rem}})$
8: \hspace{1cm} $\mathcal{M}^{\text{new}} \leftarrow \langle V_{h}^{\text{new}}, V_{d}^{\text{new}} \rangle$
9: \hspace{1cm} return $\mathcal{M}^{\text{new}}$
10: end function
Extensibility - RHEEM

RHEEM brings an additional level of abstraction

- Data quanta
- Platform agnostic data transformation operators (RHEEM plans)

When a new platform is added

- New execution operators
- Their mappings to RHEEM operators
- Data quantum specification
- Communication channels (at least one)
Extensibility - Myria

When a new platform is added:

- An AST describing the API / query language supported
- rewrite rules / mappings of logical algebra into AST
- rule ordering
- set of administrative functions (querying the catalog, issuing a query, extracting results)
RHEEM cost model learner

Operator Cost

Resource Cost

Resource Utilization

Resource Unit Cost

Parameters (automatically learned)

Selectivities (computed or provided)

Hardware Specs. (provided)

#cores=4 cpuMhz=2700...
RHEEM cost model learner (cont’d)

Parameters for a given operator and resource:

- $\alpha$ : number of required CPU cycles for each input data quantum in operator
- $\beta$ : number of required CPU cycles for each input data quantum in UDF
- $\gamma$ : fixed overhead for the operator start-up / scheduling

Logs used to learn these parameters: the cost of individual execution operators is modeled as a regression problem.

- difficulty: in logs, runtimes of stages (not individual operators)
- execution stage: $\{(o_1;C_1); (o_2;C_2); ... ; (o_n;C_n)\}; t$
- $f_i(x,C_i)$ total cost function for executing operator $o_i$
- finding $x_{min} = \text{argmin}_x \text{loss}(t, \Sigma_{i=1}^n f_i(x, C_i))$
- Genetic algorithm to find $x_{min}$
Data movement in RHEEM

Channel conversion graph (CCG)
Data movement in RHEEM

CCG: the space of possible communication steps

- vertices: data structure types (communication channels)
- edges: conversions from one structure to another (conversion operators)

Finding the most efficient communication path among execution operators: a new graph problem of finding a minimum conversion tree (similar to the Group Steiner Tree - GST problem).

- NP-hard problem, however, exp. time algorithm performs well in practice
Data movement in Myria - PipeGen

- PipeGen automatically enables optimized data transfer between arbitrary pairs of database systems
- Not dealing with schema matching / focus on “mechanics” of data movement
- Relies on DBMS capacity to ingest / export data to / from file system (CSV, JSON)
- Requires as input the DBMSs source code, unit tests exercising import / export
- Replaces that functionality with highly optimized version that sends data over a network socket
Outline

- Motivation and multiple model examples (30’)
- Theoretical foundations (30’)
- Multi-model data storage (25’)
- Questions and discussion (5’)

Session break

- Multi-model data query languages (10’)
- Multi-model query processing (10’)
- Overview on tightly integrated polystores (20’)
- Query processing in tightly integrated polystores (15’)
- Advanced aspects of tightly integrated polystores (15’)
- Comparison of multi-model databases and tightly integrated polystores (5’)
- Open problems and challenges (10’)
- Questions and discussion (5’)


Comparison of multi-model databases and tightly integrated polystores
Common features

- Support for multiple data models
- Global query processing
- Cloud support
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Multi-model DBMSs</th>
<th>TIPs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Engine</strong></td>
<td>single engine, backend</td>
<td>multiple databases (native)</td>
</tr>
<tr>
<td><strong>Maturity</strong></td>
<td>lower</td>
<td>higher</td>
</tr>
<tr>
<td><strong>Usability</strong></td>
<td>Read, write and update</td>
<td>read-only</td>
</tr>
<tr>
<td><strong>Transactions</strong></td>
<td>global transaction supported</td>
<td>unsupported</td>
</tr>
<tr>
<td><strong>Holistic query optimizations</strong></td>
<td>Open problem</td>
<td>challenging</td>
</tr>
<tr>
<td><strong>Community</strong></td>
<td>industry-driven</td>
<td>academia-driven (recently)</td>
</tr>
<tr>
<td><strong>Data migration</strong></td>
<td>difficult</td>
<td>simple</td>
</tr>
</tbody>
</table>
Outline

1. Motivation and multiple model examples (30')
2. Theoretical foundations (30')
3. Multi-model data storage (25')
4. Questions and discussion (5')

Session break

1. Multi-model data query languages (10')
2. Multi-model query processing (10')
3. Overview on tightly integrated polystores (20')
4. Query processing in tightly integrated polystores (15')
5. Advanced aspects of tightly integrated polystores (15')
6. Comparison of multi-model databases and tightly integrated polystores (5')
7. Open problems and challenges (10')
8. Questions and discussion (5')
Open problems and challenges
Multi-model databases

1. Schema design and optimization
   ○ NoSQL databases - usually schemaless
     ■ But we still need to model the target domain at least roughly
     ■ e.g. an application which stores data partially in JSON, XML and relational model => a user-friendly modeling tool which enables to model both the flat relations and semi-structured hierarchical data + relationships
   ○ Schema design influences query evaluations
     ■ Relational: minimum redundancy vs. NoSQL: materialized views
     ■ Relational: normalization vs. NoSQL: de-normalization
   ○ Schema inference
     ■ Approaches for single-models need to be extended
       ● to support references amongs models
       ● to benefit from information extracted from related data with distinct models
Multi-model databases

2. Query processing and optimization
   ○ Query languages are immature
     ■ Limited expressive power, limited coverage of models, …
   ○ The best query plan for queries over multi-model data
     ■ New dynamic statistics techniques for changing schema of the data
   ○ Indexing structures defined for single models + results are combined
     ■ e.g., relational: B-tree, XML: XB-tree, graph: gIndex, …
     ■ How to index multiple data models with a single structure?
       ● To accelerate cross-model filtering and join
   ○ In-memory technologies challenge disk-based solutions
     ■ A just-in-time multi-model data structure is a challenge
Multi-model databases

3. Evolution management
   - Schema evolution + propagation of changes
     - Adaptation of data instances, queries, indexes, or even storage strategies
     - Difficult task in general
   - Smaller applications = skilled DBA
     - Error-prone, demanding
   - Intra-model - re-use of an existing solution
   - Inter-model - distinct models cover separate parts of reality interconnected using references, foreign keys, …
     - Propagation across multiple models and their connections

4. Extensibility
   - Intra-model - extending one of the models with new constructs
   - Inter-model - new constructs expressing relations between the models
   - Extra-model - adding a whole new model
Tightly integrated polystores

Many challenges: query optimization, query execution, extensibility, interfaces, cross-platform transactions, self-tuning, data placement / migration, benchmarking.

- High degree of uncertainty even in TIPs
- Transparency: do not require users to specify where to get / store data, where to run queries / subqueries
  - Explain and allow user hints
- More than ever need for automation, adaptiveness, learning on the fly
Tightly integrated polystores

Many challenges: query optimization, query execution, extensibility, interfaces, cross-platform transactions, self-tuning, data placement / migration, benchmarking.

Query optimization:

- Query-based vs. workload based optimization
- Workload driven data partitioning, indexing, controlled degree of parallelism
- Progressive optimization $\rightarrow$ control over underlying platforms
- View-based query rewriting at large scale
- Cost based reformulation under constraints at large scale
- Cost-based optimizations across-platforms: uniformization, common cost unit, normalization $\rightarrow$ hard even in tightly coupled systems
- Computation costs vs data transfer costs
Tightly integrated polystores

Many challenges: query optimization, query execution, extensibility, interfaces, cross-platform transactions, self-tuning, data placement / migration, benchmarking.

Query execution:

- Data exploration in cross-platform settings
- Efficiently support fault tolerance across platforms
- Different query semantics / data typing

Extensibility: add new platforms automatically / easily

- Data abstractions / query abstractions

Query interfaces / internal common models / foundations:

- Expressiveness vs. declarative
- Limitations of the relational “glue” (algebra, imperative vs. Datalog-based)?
More materials:


- **UniBench: Towards Benchmarking Multi-Model DBMS**

  - Collects and integrates publicly available datasets
Conclusion

- Big Data V-characteristics bring many challenges
- Variety requires concurrent storage and management of data with distinct formats
- Two sides of the same coin?
  - Multi-model databases
  - Tightly-coupled polystores
- Still there is a long journey towards robust solutions