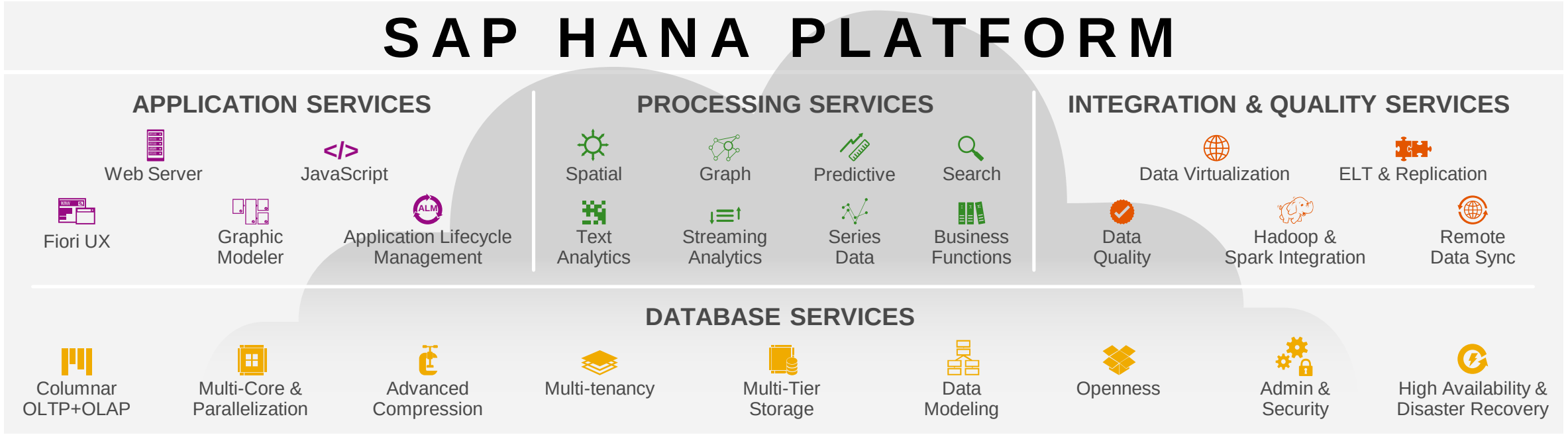
GraphScript: Implementing Complex Graph Algorithms in SAP HANA

Marcus Paradies, SAP SE
September 1, 2017

PUBLIC

SAP Run Simple

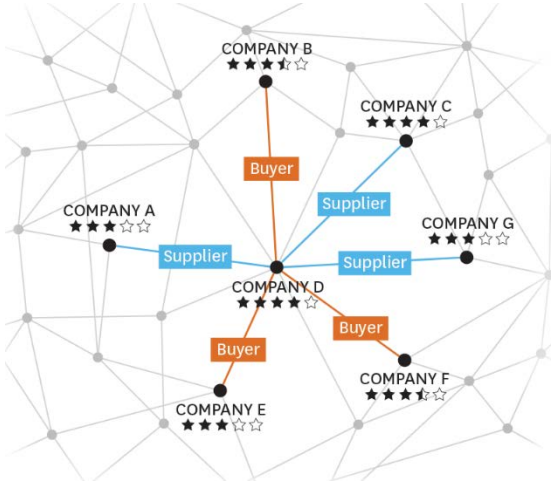
SAP HANA Overview



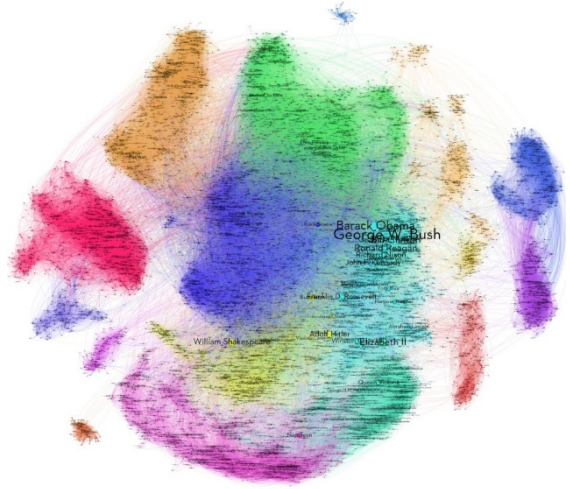
- Offers advanced analytics features for graph, text, geospatial, and machine learning directly on business data

Graph Querying Paradigms in SAP HANA

Graph Pattern Matching



Graph Analysis



Example Query

"Retrieve all suppliers of Company D"

"Compute all communities in the graph"

Language Interface

openCypher*

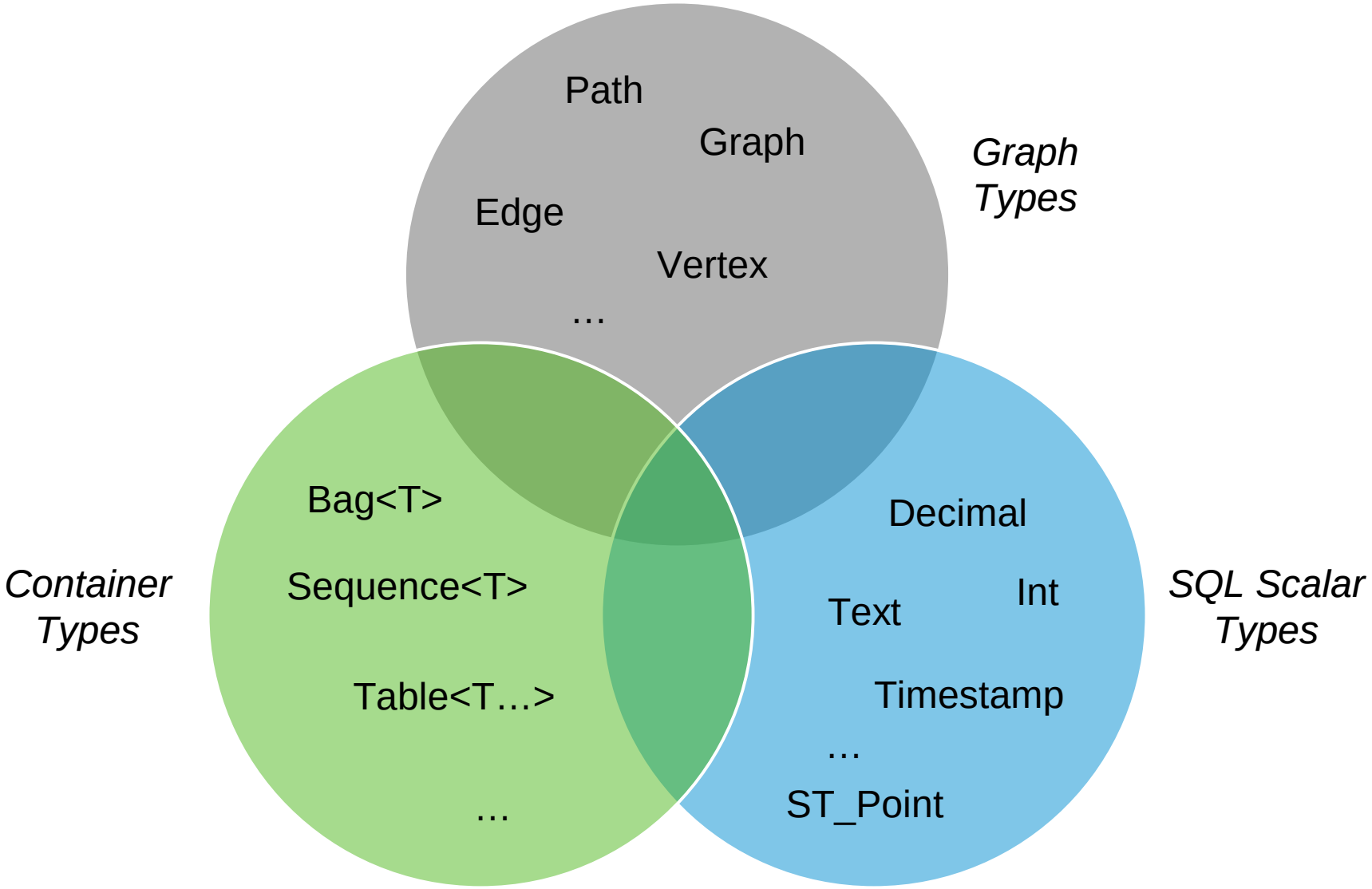
GraphScript

* Limited subset of language spec

Design Principles

Expressiveness & Simplicity	Minimality & Orthogonality	Native Graph Abstraction	Tight Integration	High Performance
• Easy-to-use for graph algorithm implementers • Support for a large variety of graph algorithm classes and workflows	• Limited but effective set of types and operations thereon • Extensibility of built-in graph operators	• Native exposure of graph-specific types • Full exposure of graph data model • Relational only for returning complex results	• Pushdown of operations to relational store • Reuse of dependency management • Reuse of resource management	• Desired performance close to hand-written code • Explicit parallelization • Effective Program Rewritings

GraphScript Type System



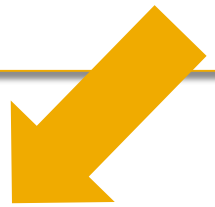
Graph Data Exposure in GraphScript

Vertex Table

```
CREATE COLUMN TABLE MYSCHEMA.VERTICES (
 ID VARCHAR(100) PRIMARY KEY,
 TYPE VARCHAR(100),
 NAME VARCHAR(100),
 TITLE VARCHAR(100)
);
```

Edge Table

```
CREATE COLUMN TABLE MYSCHEMA.EDGES (
  ID INTEGER PRIMARY KEY,
  SRC VARCHAR(100) NOT NULL
    REFERENCES MYSCHEMA.VERTICES (ID)
  TRGT VARCHAR(100) NOT NULL
    REFERENCES MYSCHEMA.VERTICES (ID)
  TYPE VARCHAR(50)
);
```



```
CREATE GRAPH WORKSPACE MYSCHEMA.MY_GRAPH
  EDGE TABLE MYSCHEMA.EDGES
    SOURCE COLUMN SRC
    TARGET COLUMN TRGT
    KEY COLUMN ID
  VERTEX TABLE MYSCHEMA.VERTICES
    KEY COLUMN ID;
```

Graph Workspace

Metadata Object

Graph Data Exposure in GraphScript /2

Vertex Table View

```
CREATE VIEW MYSCHEMA.VERTEX_VIEW AS
  SELECT * FROM MYSCHEMA.VERTICES
  WHERE TYPE = 'Person';
```

Edge Table View

```
CREATE VIEW MYSCHEMA.EDGE_VIEW AS
  SELECT * FROM MYSCHEMA.EDGES
  WHERE TYPE = 'knows';
```

```
CREATE GRAPH WORKSPACE MYSCHEMA.MY_SUBGRAPH
  EDGE TABLE MYSCHEMA.EDGE_VIEW
    SOURCE COLUMN SRC
    TARGET COLUMN TRGT
    KEY COLUMN ID
  VERTEX TABLE MYSCHEMA.VERTEX_VIEW
    KEY COLUMN ID;
```

Graph Workspace Metadata Object

A Simple GraphScript Example

```
CREATE PROCEDURE "myGraphProc"(OUT numNeighbors BIGINT)
LANGUAGE GRAPH READS SQL DATA AS
BEGIN
  Graph g = Graph("myGraph");
  ALTER g ADD TEMPORARY VERTEX ATTRIBUTE(BIGINT cnt = 0);
  FOREACH v IN Vertices(:g) {
    v.cnt = Count(Neighbors(:g, :v, 1, 3));
  }
  FOREACH v IN Vertices(:g) {
    numNeighbors += :v.cnt;
  }
END
```

Adjacency List Construction

0	1	2
1	2	5
2		
3	4	2
4	0	2
5	6	
6	2	

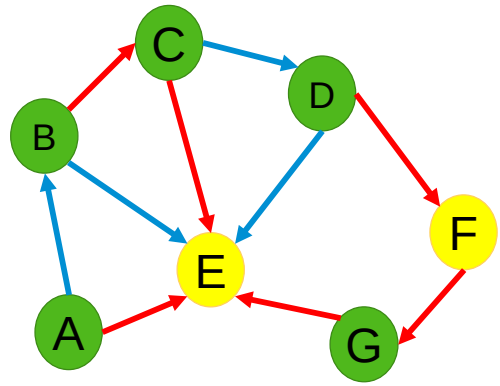
Adjacency List

0	C
1	D
2	E
3	A
4	B
5	F
6	G

Vertex Key Dictionary

Key	Color
C	green
D	green
E	yellow
A	green
B	green
F	yellow
G	green

Vertex Attributes



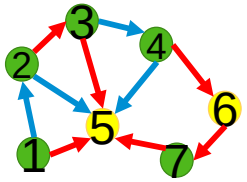
Variants:

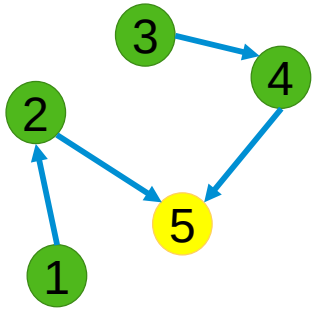
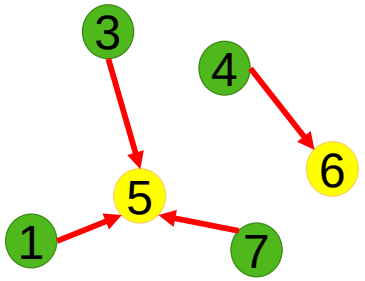
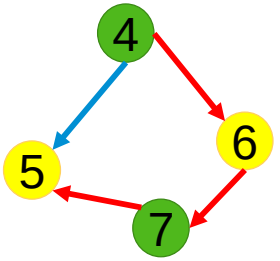
- Omit dictionary encoding for dense key domains
- Static/Dynamic/Compressed adjacency list
- Vertex/edge adjacency

Parallel Index Construction with up to 65 Mio. edge insertions/sec

Inducing Subgraphs

Data graph:



"Induce a graph over all blue edges"		`Graph g = Subgraph(:g, e IN Edges(:g) WHERE :e.color == 'blue');`
"Induce a graph over all red edges that connect a green and a yellow vertex"		`Graph g = Subgraph(:g, e IN Edges(:g) WHERE Source(:e).color == 'green' AND Target(:e).color == 'yellow' AND :e.color == 'red');`
"Induce a graph overall all vertices that are reachable from vertex 4"		`Vertex v1 = Vertex(:g, 4);` `Graph g = Subgraph(:g, v IN Vertices(:g) WHERE IS_REACHABLE(:g, :v1, :v);`

Integration with other Data Models/Scalar Types

Creation of Relational Output from GraphScript

```
Graph g = Graph("myWorkspace");
ALTER g ADD TEMPORARY VERTEX ATTRIBUTE(DOUBLE length = 0);
FOREACH v IN Vertices(:g) {
  Path p = Shortest_Path(:g, :v, Vertex(:g, 1));
  v.length = Length(:p);
}
outTab = SELECT :v.id, :v.length FOREACH v IN Vertices(:g);
```

Integration with Geospatial Processing

```
Graph g = Graph("myWorkspace");
ST_Geometry area = Vertex(:g, 'Munich').area;
Graph g1 = Subgraph(:g, v IN Vertices(:g) WHERE :v.type == 'Person'
                                          AND ST_Within(:v.location , :area));
```

Conclusion

Language Constructs

- Rich type system with native graph types
- Powerful imperative constructs

Code Generation

- Generation of low-level code against internal Graph Storage interface
- Elimination of query processing on external keys
- Pushdown of filter conditions to relational engine

Future Work

- More language extensions towards fast traversals and user-defined function invocations
- More advanced GraphScript program rewritings and optimizations

Thank you.

Contact information:

Marcus Paradies
marcus.paradies@sap.com